1995

Effects of sequencing computer-based instruction and lecture in learning function concepts of C programming language

Su Chao-Ya Tsai
Iowa State University

Effects of sequencing computer-based instruction and lecture in learning function concepts of C programming language

by

Su Chao-Ya Tsai

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Industrial Education and Technology
Major: Industrial Education and Technology

Approved:

Signature was redacted for privacy.
In Charge of Major Work

Signature was redacted for privacy.
For the Major Department

Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa
1995

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Computer programming is viewed as a major subject area in computer education because of its perceived value in promoting thinking and problem solving skills. Programming is a complex and demanding task in which one must understand a problem, design a plan to solve the problem, write that plan in computer code and debug the code (Dalbey & Linn, 1985; Lewis, 1980; Pea & Kurland, 1984). Successful programmers require knowledge of problem domain, algorithms development, programming language, error detection, and program use (Brooks, 1990; Pennington & Grabowski, 1990; Sleeman, Putnam, Baxter & Kuspa, 1986). The requirements of various kinds of knowledge make learning computer programming difficult; hence, identifying students' difficulty when they are learning to program becomes very important. Identification of difficulties provides direction which educators can use to develop effective instructional methods and learning environments for student programmers.

One pedagogical innovation computer educators made was the development of structured languages. Because of its structured design, Pascal has been the major language taught in introductory programming courses. Thus, numerous studies concerning the learning of Pascal have been done. However, recently, many educators have begun using or are considering using C language as the introductory language in

programming curriculum (Morton & Norgaard, 1993; Roberts, 1993). Unfortunately, little research has been done that addresses the learning of C language. Because of major syntax and structural differences between C and Pascal, many aspects of C as a beginning language need to be studied.

One important aspect is the manner in which C supports program segmentation. The quality of programs is highly dependent on decomposing a programming problem into subprograms (Bailie, 1991; Jeffries, Turner, Polson & Atwood, 1981; Kassab, 1989; Perry, 1992). The C language supports this modular program creation by using functions. Therefore, consideration of function concepts is extremely important in choosing learning environments for C language programmers.

One approach to building an effective learning environment is to develop computer-based instruction. Through a well designed instructional strategy, computer-based instruction can support an individualized and interactive learning environment. During the learning process, students are able to chose learning materials according to their needs at their own pace. Students are also allowed to perform tasks and interact with the computer. The problem-solving abilities which are needed in programming will as a result be enhanced during the learning process. Furthermore, combining advanced technology with computer, the instructional materials can be represented in various ways such as text, graphics, images, animation, sound, and video. Applying this advantage can promote motivation and help students understand abstract concepts in concrete ways by providing a means of relating unfamiliar material to existing knowledge. Thus, computer-based instruction has the capability to become a powerful tool that enhances the programming learning environment.

Some research suggests that if grounding structures are presented before unfa-

miliar and challenging learning material, learning can be increased (Ausubel, 1960; Mayer, 1975, 1979a, 1989). This pre-instructional approach helps learners to organize new material and to integrate it with their existing knowledge. In contrast, the post-instructional approach that related information should be presented following the instruction in order to reinforce and integrate acquired knowledge is also claimed. Several studies have been done in programming learning areas that concern these two approaches (Mayer, 1976; Mayer & Bromage, 1980; Righi, 1991). Studies conducted by Mayer and his associate reported that the pre-instructional group performed better on tasks utilizing problem-solving skills while the post-instructional group performed better on problems requiring recall competence. The studies suggested the pre-instructional approach was an effective method in learning. In contrast, Righi's study indicated that pre-instructional method can cause confusion when learning programming, and the post-instructional method was recommended by the study. Although Righi's study may be limited by the content of material provided to students, the different viewpoints of these studies suggests that additional studies are necessary in order to clarify which method is more effective.

Statement of the Problem

Studies concerning the optimal placement of computer-based lessons in instructional sequences are lacking. Furthermore, research identifying areas where students are having difficulties in learning the C programming language are also needed if instruction is to be improved. Thus, the first problem addressed in this study was the effect a computer-based lesson had on student learning when it is placed before a formal lecture on C function topics versus after the formal lecture. The second problem

addressed in this study was to identify the kind of problems students encountered in learning function concepts using the C' programming language.

Purposes of the Study

This study compared the effectiveness for beginning programmers of a computer-based lesson used before formal instruction with the same lesson used after formal instruction. The topic of the lesson was the function concepts of the C' programming language. This study also collected and examined information about the students' programming errors and misconceptions related to function concepts in C' programs.

Research Questions

There were four research questions in this study:

1. What kind of difficulties do students encounter when they learn the concepts of functions in C' programming?

2. Are there any differences in student achievement on learning function concepts in C' programming when a computer-based lesson is placed before, in contrast to after the formal instruction?

3. Does students' prior knowledge in programming affect their learning on function concepts in C' programming?

4. Is there any relationship between instructional sequence and students' prior knowledge in programming on student achievement in learning the C' function concepts?

Hypotheses

This study consisted of SEQUENCE1 and SEQUENCE2 treatments. Students in the SEQUENCE1 treatment worked on the computer-based lesson before the formal instruction. Students in the SEQUENCE2 treatment used the computer-based lesson after the lecture. The dependent variable for the study was the posttest scores of students. The independent variables selected for the study were the instructional sequence, the students' prior knowledge in programming, and the interaction of the above two independent variables. Students' attitudes toward taking the programming course, number of prior computer courses, and computer ownership are control variables (covariates) in the study. Table 1.1 shows the variables for this study.

Table 1.1: Variables in the study

Label	Description	Type	Use
Y	Posttest	Continuous	Dependent
X_1	Instructional Sequence (Treatment)	Categorical	Independent
X_2	Prior Knowledge in Programming (Pretest)	Continuous	Independent
X_3	Students' Attitudes	Continuous	Control
X_4	Number of Prior Computer Courses	Continuous	Control
X_5	Computer Ownership	Categorical	Control
X_1X_2	Interaction of X_1 & X_2	Continuous	Independent

For the purposes of this study three hypotheses corresponding to the last three research questions were formulated.

1. There is no significant contribution of the treatment variable to the prediction of posttest score variance among the subjects.

   $H_0 : \beta_1 = 0$

The comparison of a full model and a test model described below is used to test this hypothesis.

Full Model: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_1 X_2 + \varepsilon$

Test Model: $Y = \beta_0 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_1 X_2 + \varepsilon$

2. There is no significant portion of posttest score variance explained by pretest score variance.

   $H_0 : \beta_2 = 0$

   The comparison of the full model and a test model described below is used to test this hypothesis.

   Full Model: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_1 X_2 + \varepsilon$

   Test Model: $Y = \beta_0 + \beta_1 X_1 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_1 X_2 + \varepsilon$

3. There is no significant interaction between the instructional sequence treatments and student prior knowledge as measured by the contribution of interaction to the prediction of posttest score variance.

   $H_0 : \beta_6 = 0$

   The comparison of the full model and a test model described below is used to test this hypothesis.

   Full Model: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_1 X_2 + \varepsilon$

   Test Model: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \varepsilon$

Assumptions

Four assumptions were foundational to the study:

1. The students involved in this study responded as honestly and as accurately as possible to the questions asked.
2. The instruments used in this study had satisfactory reliability, validity and sensitivity to the experimental effects.
3. There were no systematic factors occurring during the study which affected the experimental results.
4. The experimental time was adequate to produce a measurable effect.

Delimitations

The study was delimited by the follows:

1. The sample for this study was selected from Industrial Education and Technology students at Iowa State University. Results may only generalize to similar populations.
2. The research was restricted to the examination of student difficulties in learning function concepts of the C programming language.
3. The unit of observation in this study was the individual students. To the degree that students may have shared information among themselves, thus reducing the independence of treatment effects, may affect the validity of the experimental design.

4. The development of computer-based instruction was a complex, difficult, expensive, and time-consuming process requiring much effort and support. The limitations of developmental time and expertise may therefore have limited the potential effectiveness of this method.

Definition of Terms

C programming language: A high level computer language with powerful features used in education and industry.

Student difficulties: Misconceptions, programming errors, and the effort required that students experience when they learn programming.

Instructional sequence: The order of the computer-based lesson and the formal instruction used in this study. There are two different instructional sequences in the study. One is to present a computer-based lesson prior to formal instruction, and the other after.

Decomposition: The process of breaking up a problem into simple parts.

Arguments: Variables that received values passed to a function.

Lesson function: The name of a computer-based lesson used in the study.

Project Vincent: A computer network used at Iowa State University.

Advance organizer: Introductory material presented prior to unfamiliar and challenging learning material to help learners integrate the incoming information into their existing cognitive structure.

Mental models: Learners' internal representations that are used in predicting, planning, and interpretation of the system behavior (van der Veer, 1993).

Notional machine: The idealized model of the computer implied by the constructs of the programming language (du Boulay, O'Shea & Monk, 1981).

CHAPTER 2. LITERATURE REVIEW

The literature review is based on the cognitive aspects of programming and educational concepts of learning and instruction. It is organized into five sections: (1) cognitive activities of programming, (2) student difficulties in programming, (3) related learning theory and instructional design, (4) research on pre-instructional and post-instructional methods, and (5) summary of literature review.

Cognitive Activities of Programming

A program is a set of instructions that directs a computer to perform operations. Students learn to develop computer program in a programming course which is also believed to help students develop thinking and problem-solving skills. Computer programming is a highly complex process that can be broken down into four parts: specification, design, coding, and debugging (Dalbey & Linn, 1985; Lewis, 1980; Pea & Kurland, 1984). Those activities have multiple interconnections which make them difficult to separate. (Pennington & Grabowski, 1990). Thus, expert programmers usually engage in these activities recursively until their programs work correctly.

The initial phase of programming is specification. Specification is understanding and analyzing a problem. It involves evaluating the given information, recognizing the overall goal, and outlining the operations for applying the given information in

order to achieve the goal (Brooks, 1983). In other words, one must identify the input, the output, and any other relevant information and determine how the information can be employed in a program.

In addition, specification requires not only knowledge of a computer language but also knowledge of the subject area in which the problem lies, such as mathematics, electronics, or mechanics. Therefore, students often encounter difficulties in identifying a problem because of deficiencies in their understanding of the required background knowledge or domain knowledge. The deficiencies may involve conceptual misunderstandings, incomplete information, confusing notation, or unclear conditions of application (Eylon & Linn, 1988; Simplicio-Filho, 1993). Even for simple programming problems, teaching one how to understand a problem is difficult (Rogalski & Samurçay, 1993).

The second phase of the computer programming process is design. Design is a complex and important task in programming. Design involves finding a problem solution through a structured detailed plan. One aspect of design is problem decomposition.

Problem decomposition is a crucial technique for solving complex problems (Lewis, 1980; Polya, 1957). The philosophy of problem decomposition is breaking down a complex problem into manageable and independent subproblems. The reduced size and complexity of the problem minimizes the cognitive load and makes a program easy to read, write, modify, and debug (Brooks, 1977).

A study conducted by Jeffries et al. (1981) showed that expert programmers were skilled at decomposition, but novice programmers had difficulty breaking down problems into appropriate subproblems. Evidence also revealed that decomposition

is a determining factor in programming success. Further, expert programmers were found to spend more time in designing their problem solutions, whereas novices usually go to computer directly without planning (Jeffries et al., 1981).

Two types of problem decomposition exist: top-down design and bottom-up design. Top-down design starts at the program's general goal and proceeds toward more detailed code generation. Bottom-up design begins at detailed code generation and proceeds toward the program's general goal. Top-down design is advocated by many educators in the programming area (Jeffries et al., 1981; Kassab, 1989; Rist, 1986). However, in some situations top-down design does not work well because the nature of design strategies is usually influenced by the problem characteristics or programmers' experiences (Ratcliff & Siddiqi, 1985; Sumiga, 1993). Novice programmers often use the bottom-up approach (Rogalski & Samurçay, 1993). It seems that both top-down and bottom-up approaches are necessary and need conveying to students.

A review of programming research found that design is the most important cognitive activity in programming (Dalbey & Linn, 1985). Unfortunately, design is usually neglected by educators because of the difficulty in presenting the required knowledge and demonstrating the process (Ferguson & Henderson, 1987). Specifically, design is ignored because of the difficulty in relating the interactions among design and specification, coding, and debugging (Pennington & Grabowski, 1990).

The third phase of programming is coding. Coding involves translating the problem solution into a specific programming language. According to Fay and Mayer (1988), the cognitive chain of knowledge must be acquired for successful program writing. The knowledge domains in the chain are syntactic, semantic, and problem solving knowledge.

Syntactic knowledge of a computer language consists of the language elements (e.g., reserved words, variables, operators, and punctuation marks), and the rules of combining these elements into a statement or a program (Fay & Mayer, 1988). Semantic knowledge of programming includes understanding the logical meaning of computer language syntax and operations of the computer (Hoc & Nguyen-Xuan, 1990; Shneiderman & Mayer, 1979).

Syntactic knowledge of programming is the basic requirement for successful programming, and semantic knowledge of programming must be obtained for effective programming (Fay & Mayer, 1988). Syntactic knowledge can be increased by practice. Semantic knowledge can be enhanced if programmers have a clear "mental model" of the computer (Hoc & Nguyen-Xuan, 1990; van der Veer, 1993). Research showed that expert programmers have a more well-constructed mental representation of the computer than novice programmers (Hoc, 1977).

In addition, studies have suggested that learning to program involves organizing group data into reusable "chunks" (Adelson & Soloway, 1985; Linn & Clancy, 1992; Shneiderman & Mayer, 1979). Those chunks can be subprograms such as procedures or functions which perform specific tasks. Expert programmers usually excel on developing chunks and modifying or reusing them when dealing with new problems where novices usually fail to do so (Linn & Dalbey, 1985).

When writing a program, abstract thinking ability is required. Programmers need to think according to the operations of a computer, not everyday life experiences. Actually, coding cannot be separated clearly from design activity (Pennington & Grabowski, 1990). Thus, some concepts of design task must be taken into account during coding.

The final phase of computer programming is debugging. Debugging involves identifying, locating, and correcting errors in computer programs. It is a diagnostic activity requiring program comprehension and formal reasoning. Debugging is a difficult and time consuming task. Research showed that programmers spend most of their programming time in debugging (Allwood & Bjorhag, 1990).

Programming bugs can be classified into syntax errors, semantic errors, and logic errors. Usually, the syntactic errors are easy to detect and correct with the aid of error messages provided by the computer, but the semantic errors and logic errors are hard to identify due to the requirement of problem-solving skills (Allwood & Bjorhag, 1990; Gugerty & Olson, 1986). Furthermore, these errors often result from a misconception on the part of the novice programmer which must be corrected before the error can be eliminated.

Expert programmers commit about the same number of errors as novices (Youngs, 1974). Expert programmers however have a superior approach in identifying errors and are able to correct them more quickly. Novice programmers, on the other hand, make more incorrect identifications of errors thus creating even more new errors during debugging (Gugerty & Olson, 1986). Furthermore, novices often use trial-and-error approaches to detect program errors and are not always able to correct the bugs that they had found (Kessler & Anderson, 1986).

With rich knowledge and experience, the expert programmers use several approaches to find program errors. The approaches used include evaluating the program, analyzing error messages, using output information, testing the internal program states, and recalling previous experiences (Gugerty & Olson, 1986). Although debugging strategies can be taught by instruction, students must acquire the skills

on their own.

As described above, different cognitive knowledge and skills are needed in different programming activities. The requirements of both breadth and depth knowledge and skills increases the difficulty of programming.

Student Difficulties in Programming

Perkins, Hancock, Hobbs, Martin, and Simmons (1986) observed that students who succeeded at learning to program did not give up when they encountered problems. They viewed problems as exciting challenges and tried to work through the problems. On the other hand, students who were less successful at learning to program felt the problems as a source of frustration and gave up when confronted with difficulties. Students should be taught that facing difficulties and making mistakes are part of the programming process, and there is nothing to be afraid of. It is necessary for students to build their self-confidence and get additional support from the learning environment.

Research has been conducted investigating program errors that students most often made and the reasons why they created those bugs (Pea, 1986; Spohrer & Soloway, 1986a). Studies have also investigated programming difficulties in fundamental language concepts such as input, output, assignment, conditional statements, and looping (du Boulay, 1986; Putnam, Sleeman, Baxter & Kuspa, 1986; Soloway, Bonar & Ehrlich, 1983). These difficulties have been studied by using programs which do not contain subprograms.

Subprograms which can be implemented as procedures or functions in the Pascal language or functions in the C language are an important component of a computer

language. For the experienced programmer subprograms decrease the program complexity and are beneficial in the creation, comprehension, and debugging of a program. Unfortunately, little research has addressed the problems that beginning students encounter when using subprograms (Carrasquel, Roberts & Pane, 1989; Fleury, 1991a, 1991b).

Carrasquel et al. (1989) observed students in Pascal programming and found that students had difficulties dealing with the concepts of procedures or functions. They reported that students had trouble differentiating local and global variables, declared variables in wrong places, had difficulty deciding the type of parameters, passed unnecessary or wrong parameters, and created similar subprograms.

Fleury's study (1991b) supported these observations. In addition, Fleury found that beginning students were confused by subprograms that call other subprograms and students lost the trace of program flow. She concluded that the computer language rules may have contributed to students' partial knowledge, but some incorrect or incomplete conceptions of the rules were a major source of students' misunderstanding.

Difficulties occur because novice students do have not a clear "mental model" about the semantics of the computer language. In addition, students may struggle with the computing system they use. Rogalski and Samurçay (1990) stated that:

> Novices not only have problems with the representation of the machine underlying programming languages but also encounter difficulties with the computing system they are working with. Beginners need to differentiate which elements of this system belong to the language, and which are system entities. (p. 165)

A study suggested that many program errors were not due to misunderstanding

about the semantics of computer language but due to difficulties of "putting the pieces of a program together" (Spohrer & Soloway, 1986b). Joni and Soloway (1986) reported that many students wrote working programs with poorly constructed codes. These studies revealed that students have difficulties in designing a plan.

Furthermore, novice students usually have difficulties in tracing and debugging computer programs. Putnam et al. (1986) reported that in the process of debugging, novice students focused on a few statements, reasoned the program tasks based on small segments, and had problems keeping track of the data flow. Students seemed to assume that the computer had human abilities to interpret the programs (Pea, 1986; Putnam et al., 1986).

Researchers have indicated that understanding the program is an important factor in debugging (Gugerty & Olson, 1986; Nanja & Cook, 1987; Pennington & Grabowski, 1990). In the debugging process, experienced students tried to understand the whole program first; whereas, beginning students spent less time in understanding the program and were anxious to search program errors (Nanja & Cook, 1987). The study revealed beginning students do not have enough experience to recognize the importance of understanding the program, or they lack the required knowledge to comprehend the program.

Related Learning Theory and Instructional Design

The intention of learning theories are to provide a framework for instructional design in order to overcome learning difficulties and facilitate learning. Learning theories have shifted from a behavioral to a cognitive point of view. Behavioral learning theories view learning as a change in behavior or performance through experience

or practice (Jonassen, 1991). Behavioral approaches provide an appropriate environment in which reinforcement is utilized for correct or proper behavior. What learners do is the major focus of behavioral theories. In contrast, cognitive learning theories emphasize the construction of knowledge and the changes in understanding rather than changes in behavior (Shuell, 1986). Cognitive approaches are to encourage learners to construct their own knowledge. What learners know and how they process information is the focus of cognitive theories.

Shuell (1986, 1992) defined learning as an active, constructive, cumulative, self-regulated and goal-oriented process. From this standpoint, learning is an active construction process. It is learners who take responsibilities during learning. They need to build a mental representation of the new material and integrate the new information with knowledge already existing in their memory.

Meaningful learning has been advocated by researchers (Mayer, 1975, 1976, 1981, 1989; Shuell, 1986, 1992). This type of learning is especially useful in learning complex or abstract concepts such as domain knowledge in computer programming, science, and mathematics. Meaningful learning builds both internal and external connections (Mayer, 1989). Internal connections require the establishment of the relationships among concepts of new materials, and external connections refers to the links between unfamiliar incoming material and the student's existing cognitive structure (Mayer, 1975).

Mayer (1981) proposed a meaningful learning model to assist less experienced programmers to learn computer programming. Three steps, "reception", "availability", and "activation", are involved in the model.

During the reception step, the incoming material must be received by the stu-

dent's short-term memory. If the student does not pay attention to the material, the transfer of new information from outside to short-term memory cannot occur. Therefore, the new material and students' characteristics such as value, motivation, interest, expectation and emotion should be taken into consideration in this stage.

In the availability step, the student must retain required knowledge in long-term memory for incorporation with new information. Thus, what the student already knows is a important factor that influences learning. If students do not accumulate required knowledge in long-term memory, additional information is incomprehensible and must be learned by rote.

Finally, in the activation step, the student incorporates the required knowledge stored in the long-term memory with the new information; hence, building the knowledge structure. If students possess required knowledge but do not recognize the relationship between new information and previous knowledge, the transfer of old information from long-term memory to short-term memory will not take place. As a result, the new material may be added to memory in a isolated way.

Learning is a complex and dynamic process. The purpose of instruction is to maximize the individual's potential and learning. Not all learning is the same. Effective instruction needs to accommodate different kinds of learning and make sure students engage in proper learning material and activities (Shuell, 1986).

When the body of knowledge is well-structured, the direct instruction approach appears to be most successful. In contrast, when the body of knowledge is ill-structured, the instruction should involve less direction and provide more activities (Dembo, 1994; Eylon & Linn, 1988).

Programming requires a vast knowledge. Successful programming requires knowl-

edge of problem comprehension, design strategies, a computer language, and debugging. The large body of knowledge contains both declarative and procedural knowledge (Pennington & Grabowski, 1990). The declarative knowledge of programming such as the features of the computer language is well-structured and explicit. The procedural knowledge of programming such as comprehending programs, writing programs, design strategies, and debugging skills are ill-structured and implicit.

Learning to program by example was suggested by Pirolli and Anderson (1985). They found that in the early stage of programming, students relied heavily on examples available to them. Lieberman (1986) showed that using concrete examples to visualize the operations of a computer can clearly represent ideas and aid students in learning abstract concepts. Recently, Segal and Ahmad (1993) reported that many students focused their attention on examples, but this created some misconceptions for students due to a lack of full understanding. Because limited examples can not convey the whole complex knowledge, they suggested that the learning material should combine definitions, rules, and explanation with examples.

Sloane and Linn (1988) suggested that well-developed instructional material assisted students in integration and application of their knowledge. In addition, their study reported that direct instruction and relevant feedback enhanced programming analyzing and modifying, and problem-solving practice improved student performance in program writing. Direct instruction mainly benefits medium and low ability students, because these students lack problem-solving skills (Linn & Dalbey, 1985).

Another study conducted by Husic, Linn and Sloane (1989) suggested that instruction should provide students opportunities for problem-solving and support them

in learning the skill of monitoring their own progresses. Moreover, they concluded that when instruction emphasized direct instruction, students lost chances to practice procedural skills. If instruction focuses on self-discovery, students may need to spend more time but make little progress. Mayer (1988) suggested that "a hands-on discovery environment should be complemented with direct instruction" (p. 5).

Educators have been challenged with teaching procedural knowledge. Even experts who are experienced in employing this knowledge have difficulty explaining the process. Eylon and Linn (1988) suggested that procedural knowledge can not be learned effectively from direct instruction, the best way for students to gain this ability is offering them an environment which permits practice.

"Manipulative models" have been constructed to facilitate the acquisition of procedural skills (Hooper & Thomas, 1990; Upah & Thomas, 1993). The manipulative models require students to perform tasks and observe the internal computer status when commands are executed. This approach attempts to help students build their mental model about how the computer works.

The study constructed by Hooper and Thomas (1990) used the "notional machine" with both "visible" and "hidden" approaches. The study revealed that the manipulative model influences students to choose better algorithms and significantly improves the acquisition of procedural knowledge.

Manipulative models used by Upah and Thomas (1993) required students to write code segments in Pascal and perform looping tasks. Results of their study showed that manipulative models increase students' understanding of abstract and difficult concepts and improve students' abilities in solving unfamiliar programming problems.

These two studies both used simulation type computer-based instruction by utilizing the computer as a learning device. Kulik and Kulik (1991) reported that computer-based instruction positively motivates students and assists their learning. The computer also has capabilities to support knowledge representation in multiple ways. This strength provides alternative views of the same knowledge and helps students understand the complex concepts thus decreasing misconceptions (Confrey, 1990). In addition, computer learning environments provide individualization and interaction and are an ideal educational agent in passing on general problem solving strategies (Shuell, 1992).

Many programming learning environments therefore are built based on the computer. These environments emphasize computer language feature (Anderson & Reiser, 1985), support design or debugging activities (Bonar & Cunningham, 1988; Ferguson & Henderson, 1987; Johnson, 1990; Spohrer, 1992), provide a notional machine (du Boulay , O'Shea & Monk, 1981; Hooper & Thomas, 1990; Ramadhan & du Boulay, 1993), and support overall programming activities (Brusilovsky, 1993).

Research on Pre-Instructional and Post-Instructional Methods

The pre-instructional method introduces relevant information to learners before instruction. The post-instructional method presents related information to students after instruction. Both methods are broadly used as instructional strategies. The purpose of the pre-instructional method is to bridge the gap between learners' existing knowledge and new materials. On the other hand, the post-instructional method provides learners opportunities to enhance and extend the new materials.

The concept of the pre-instructional method was proposed by Ausubel (1960)

in terms of "advance organizers". He hypothesized that introducing the general, abstract, and inclusive "relevant subsuming concepts" prior to instruction serves to provide the learner with "appropriate anchoring ideas" that are required for comprehension of new material. Based on this belief, several studies showed that advance organizers have positive effects in facilitating learning and retention (Ausubel, 1960; Ausubel & Fitzgerald, 1961, Ausubel & Youssef, 1963).

Mayer (1979b) reinterpreted Ausubel's subsumption theory and conducted his own assimilation theory. He suggested that an advance organizer which provides the learner with a familiar and rich set of prior experiences will facilitate the interaction between short term and long term memory and aid learners in understanding and organizing new information. Based on a series of studies conducted by Mayer and his associate (Mayer, 1975, 1976, 1979a, 1979b, 1981; Mayer & Bromage 1980), three general evidences were reported. First, advance organizers perform better in learning abstract and unfamiliar information than familiar and concrete materials. Second, as compared with high-ability learners, low-ability students gain more benefit from advance organizers. Third, advance organizers have a superior impact on high thinking skills than on simple memorization.

In a review of 32 advance organizer studies, Barnes and Clawson (1975) reported that only 12 of 32 studies had significant results. Therefore, they stated that "advance organizers, as presently constructed, generally do not facilitate learning" (p. 651). This study was strongly criticized on its methodology (Mayer, 1979a). Some researchers continue to conduct additional reviews and report that advance organizer facilitates both learning and retention (Luiten, Ames, & Ackerson, 1980; Stone, 1983).

Recently, research in various academic areas showed no consistent evidence for the success of advance organizers (Carnes, Lindbeck & Griffin, 1987; Doyle, 1986; Krahn & Blanchaer, 1986). These conflicting results may occur due to the lack of a precise definition of advance organizers, the difficulty in constructing an advance organizer, and the challenge in controlling the educational variables. The instructional materials used by researchers so far include text, graphs, videotapes, tutorial type computer-based instruction, and simulation type computer-based instruction. Furthermore, the content of advance organizers involves abstract information defined by Ausubel (1960) or concrete model suggested by Mayer (1976, 1981, 1989). Apparently, a universal accepted definition and construction of advance organizers are needed.

The post-instructional method can be viewed as offering "post organizers" to learners. Unlike advance organizers, post organizers do not have a solid cognitive framework supporting its effectiveness. Harrington (1968) and Bauman, Glass and Harrington (1969) found that post organizers are helpful in promoting learning (cited in Peterson, Lovett, Thomas, & Bright, 1973). The study by Callihan and Bell (1978) supported this point and reported that post organizers have a better effect on immediate mathematical learning than an advance organizer. More than likely, a post organize can serve to help review and retrieve learners' past learning materials.

Several studies involve both advance and post organizers (Alexander, Frankiewicz & Williams, 1979; Brant, Hooper & Sugrue, 1991; Mayer, 1976; Mayer & Bromage, 1980; Righi, 1991). The study performed by Alexander et al. suggested that advance and post organizers are both useful in facilitating learning. The research conducted by Brant et al. showed that students using an advance organizer performed sig-

nificantly better than students receiving a post organizer and students receiving no treatment, while students using the post organizer did not have a significantly higher posttest mean score than students who did not receive the organizer. Mayer's study (1976) related to programming language reported that subjects given an organizer before learning performed better on incorporating new material into existing knowledge; whereas, subjects who received the organizer after the learning were good at connecting the details among the new materials.

In the study of Mayer and Bromage (1980), an organizer served to concretely describe the main location of the computer in terms of input numbers, memory space, the sequencing execution and output results. The learning material in the study consisted of a seven-page manual that explained a simplified version of BASIC or FORTRAN computer language which contained READ, WRITE, CALCULATE, EQUALS, GO TO, IF and STOP statements. Sixty university students participated in the study and were assigned to before or after groups, where the before group received an organizer prior to learning, and the after group was given an organizer after the learning. Their results showed that the before group was superior in far transfer tasks which required applying knowledge to solve an unfamiliar problem, and the after group was better on near transfer tasks which were similar to the material that had been learned recently. They concluded that presenting an organizer before the instruction may create a meaningful learning condition for future learning, and giving an organizer after the instruction may delay the chance of meaningful learning.

Most recently, Righi (1991) conducted a videotaped organizer (7 min, 35 sec) to teach grade school students programming. The organizer used in the study compared the activities of a waitress and cook in a restaurant to the activities of computer

programming. The learning material was fundamental concepts of BASIC language and was arranged into two videotaped lessons. The results showed that the advance organizer did not enhance learning, but the post organizer did. Therefore, the study suggested that for learning the more difficult material such as programming, the advance organizer may cause misconceptions, and the post organizer may be more effective. Because the relevance of the organizer and the learning material was not explained in detail in the report of the study, this study might be limited by the content and short time of videotaped organizer. However, the inconsistent results from previous research indicated that further studies are needed.

Summary of Literature Review

Computer programming is a complex and dynamic cognitive process which involves understanding a problem, designing plans, writing programs and debugging programs. The need of a broad knowledge enhances the difficulty of learning programming. Difficulties that students usually encounter in programming have been investigated. Learning theories are helpful in understanding the learning process. They provide guidelines for instructional design that may assist students to overcome difficulties and maximize their learning.

In programming, different types of instruction are necessary depending on which cognitive activity students are in and what kinds of knowledge are required, The computer has potential features for providing a powerful programming learning environment. Many kinds of computer-based instruction have been developed for helping students learning to program.

When computer-based instruction is used as an organizer and is provided before

formal lecture, it offers related experiences for later instruction and increases the opportunity of meaningful learning. When computer-based instruction is placed after the formal lecture, it can strengthen learning outcomes and provide an opportunity for students to associate previously isolated concepts and apply recently acquired knowledge. However, the question regarding which instructional sequence is more effective remains open; thus, further investigation is necessary.

CHAPTER 3. METHODOLOGY

This chapter summarizes the research methodology of the study. It consists of five major sections: (1) subjects, (2) description of the computer-based lesson, (3) instruments, (4) research procedure, and (5) data analysis.

Subjects

The subjects of this study were the students of the Computer Applications class (IEDT 216) in the Department of Industrial Education and Technology at Iowa State University during the fall semester of 1994. The use of these subjects was approved by the Iowa State University Human Subjects Review Committee.

At the beginning of the semester, the subjects were asked to participate in the study voluntarily. Of the 20 volunteers initially, 15 students participated in the study. The data of the remaining five students were not included because four students dropped out from the course, and one student had many computer programming experiences and was considered as an outlier.

All 15 subjects were male industrial education technology majors. One student was a freshman, two were sophomores, ten were juniors, and two were seniors. Seven students had never taken a computer course, and eight students had taken some type of high school or college level computer course. Two students indicated they had

programming experience in BASIC, FORTRAN, and C, two in BASIC, Pascal, and FORTRAN, one in BASIC and FORTRAN, one in BASIC and Logo, two in BASIC, and two in FORTRAN. Seven students reported they had computers available for their use at home.

Description of the Computer-Based Lesson

A computer-based lesson called *lesson function* was designed to help students learn the concepts of functions in the C programming language. *Lesson function* was created by using the ABC system (Boysen, 1992, 1993, 1994) which is available on Project Vincent workstations. ABC is an object-oriented instructional system developed by Dr. Pete Boysen of the Computation Center at Iowa State University. ABC system was chosen because it supports various instructional strategies which empower the computer as a learning device and is readily available.

Lesson function was used to provide instructional activities and obtain data for the study. Computer files were maintained that recorded students' programming errors, exercises completed, questions asked, and the time in which students clicked pre-determined buttons.

The lesson provided explanative, example, and conceptual materials, as well as a communication tool. The learning materials were organized into six sections: (1) getting started, (2) basic concepts of functions, (3) functions in C programs, (4) how functions pass values, (5) scope and duration of variables, and (6) asking questions. The representation forms of the materials included textual and visual formats which were employed under different conditions and purposes.

The explanative materials of the *lesson function* contained fundamental infor-

mation such as basic guidelines, definitions, syntactic and semantic knowledge, and structural format. Three sample pages are given in Figure 3.1, Figure 3.2 , and Figure 3.3.

Figure 3.1 explained calling function, called function, program control and the relationship between each term. Figure 3.2 was shown at the same time on the computer screen to the right hand side of Figure 3.1. Figure 3.2 used a diagram to present the program flow. When a student depressed and held the mouse button on each function name, a popup message appeared describing the relationship of this function to other functions. In this example, text and graphics supported each other and represented knowledge. In Figure 3.3, a function definition was explained followed directly by the structure of the C function. Students could click on each element of the structure and view the detailed description.

Example materials provided several examples of functions and complete programs. Two sample pages can be found in Figure 3.4 and Figure 3.5. Figure 3.4 presented a simple program which prints a message on the computer screen ten times. Students were allowed to click on key statements for further information or click on the "Alternative" button to see another way to write the program which did not use a prototype. When students clicked on the "Run" button, a programming environment (Figure 3.5) was shown on the computer screen to the right hand side of Figure 3.4.

The programming environment consisted of a menu bar, message window, edit window, data input window, and error or output window. All the windows except the message window were scrolled windows. The message window was used to display related messages. The edit window allowed students to modify or write programs.

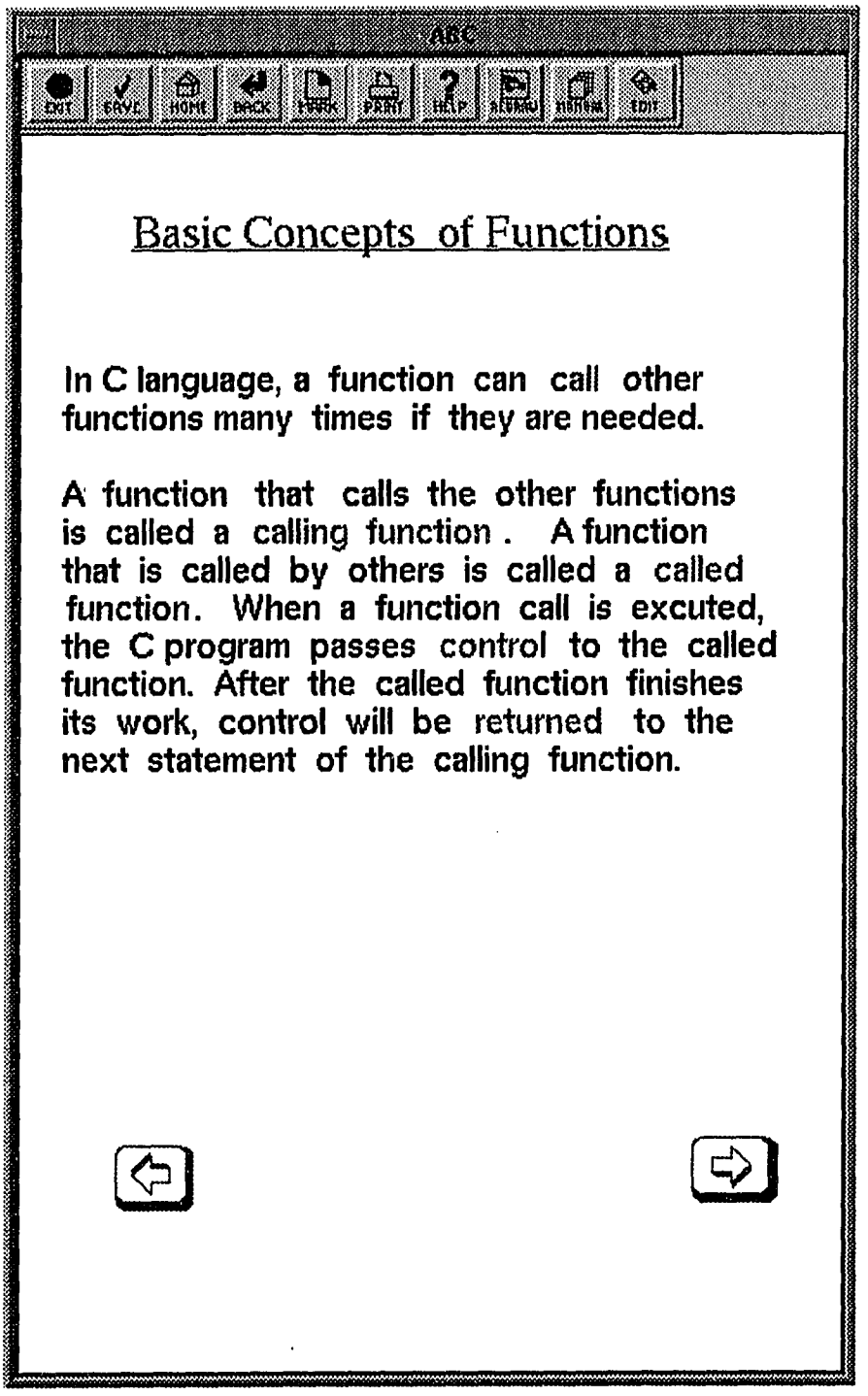


Figure 3.1: The calling function and called function

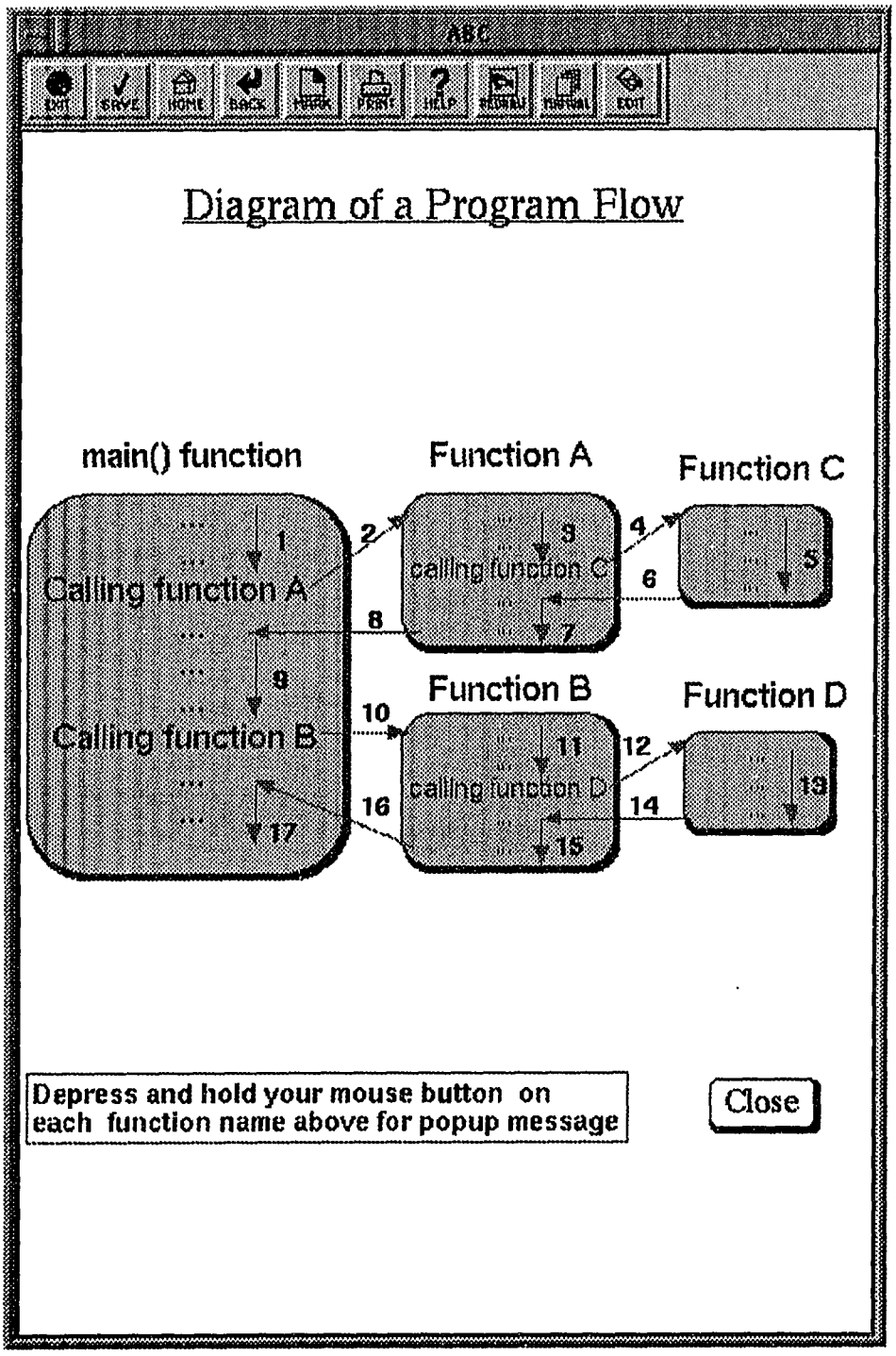


Figure 3.2: Diagram of program flow

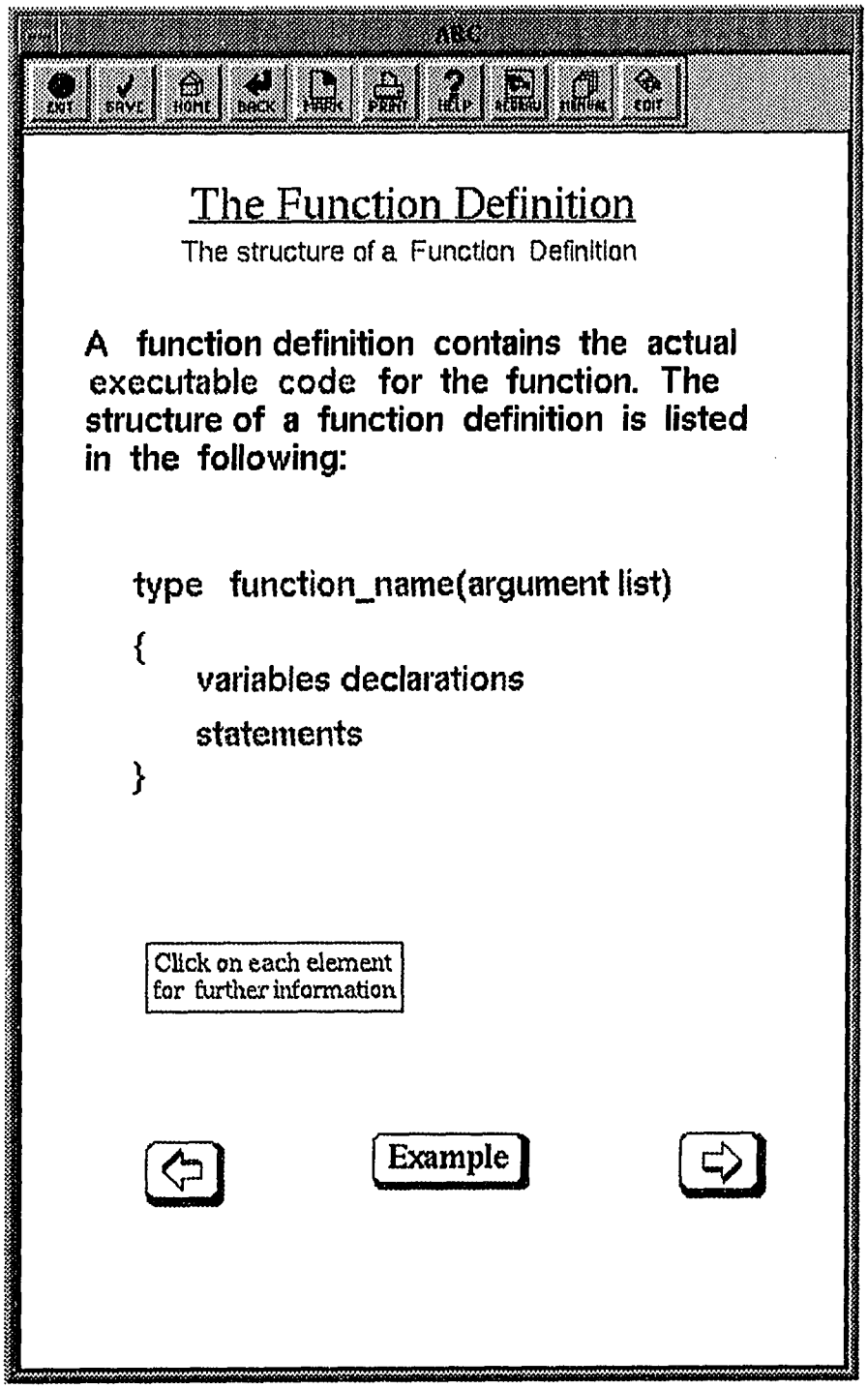


Figure 3.3: The structure of C functions

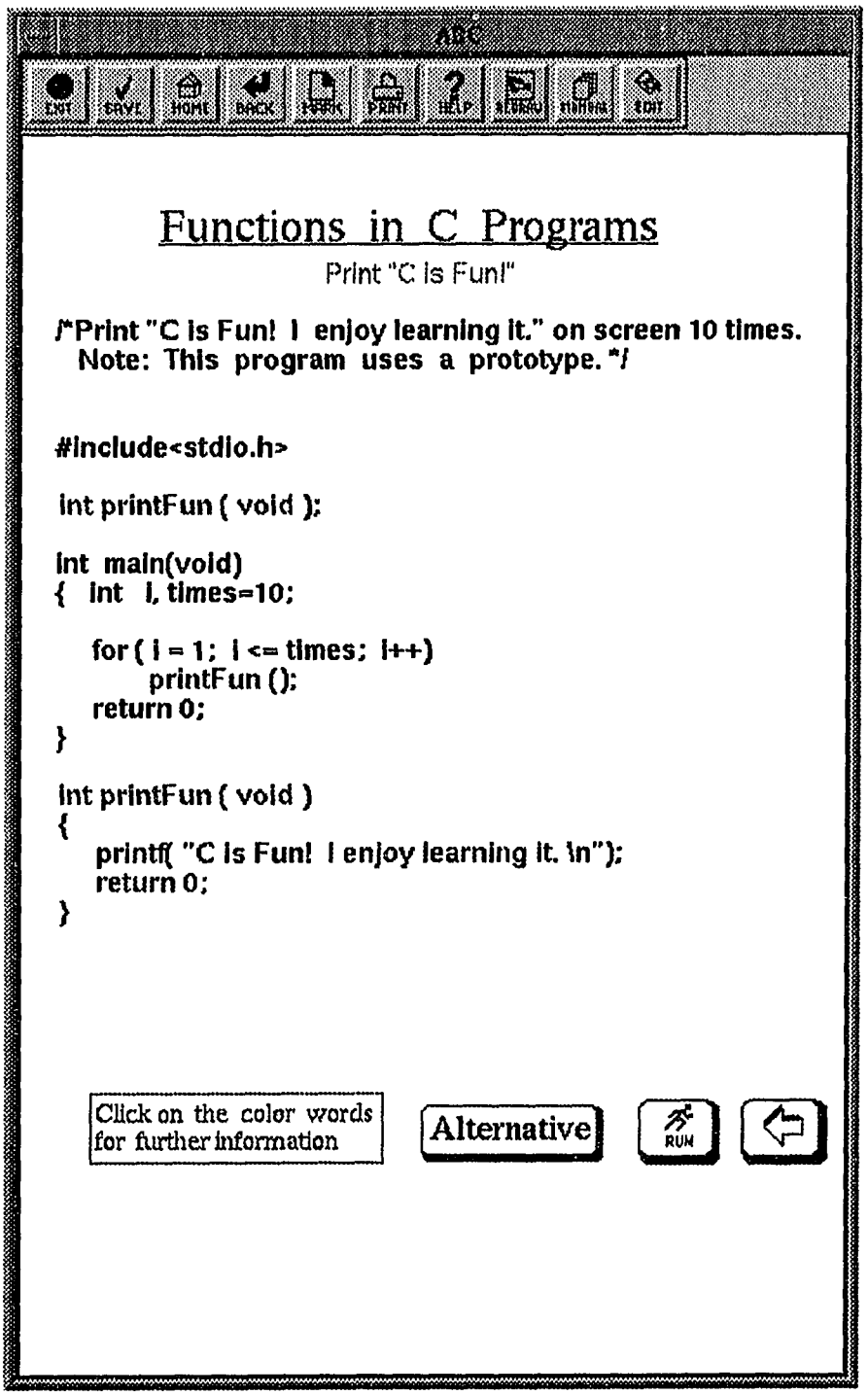


Figure 3.4: A program example

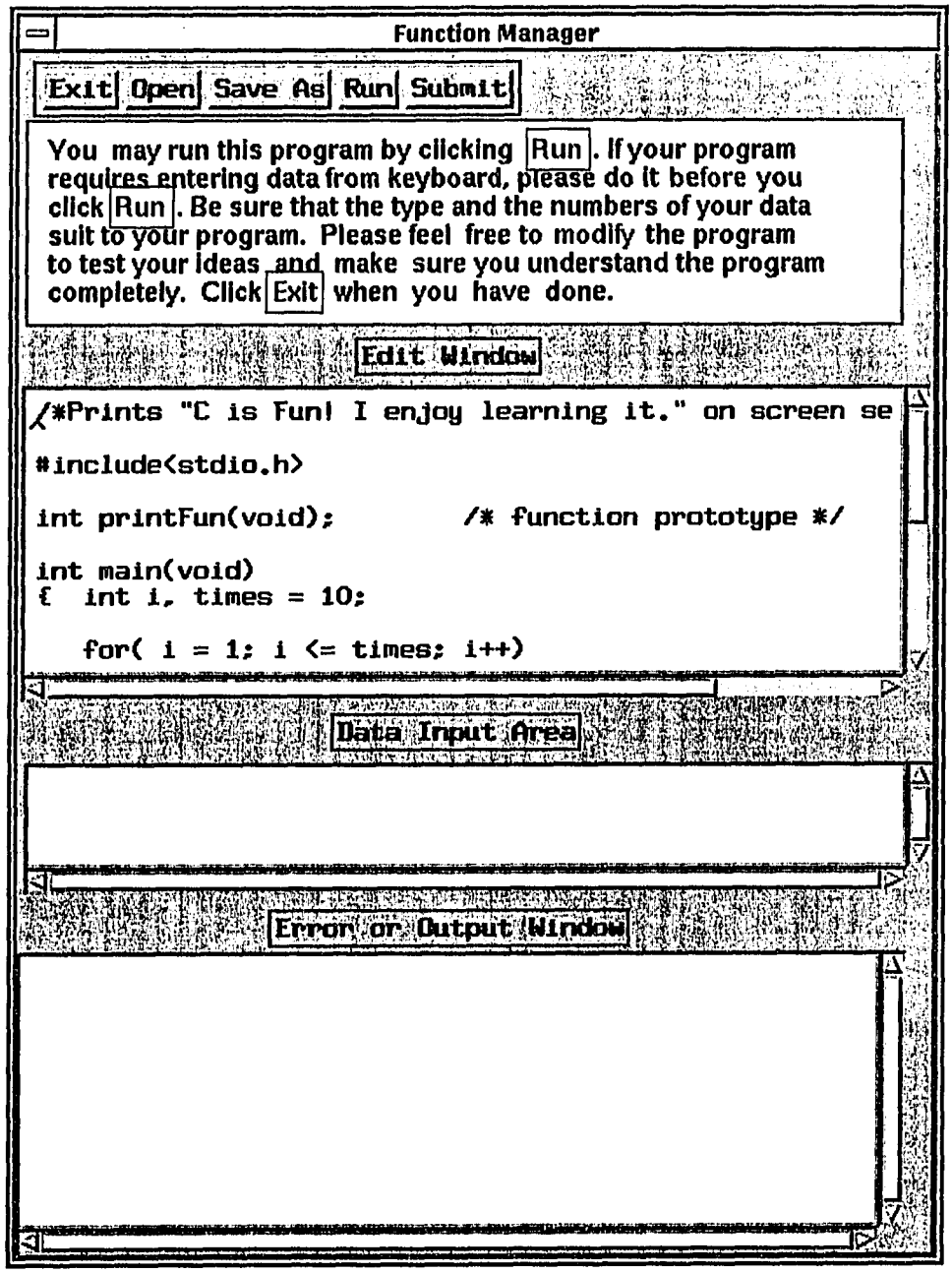


Figure 3.5: Programming environment

The data input window was used to key in data if a program contained an input statement. The error or output window displayed error messages when the program had an error or presented output results when the program was executed successfully. The menu bar consisted of "Exit", "Open", "Save As", "Run", and "Submit" buttons.

Students could exit the programming environment at any time by clicking on the "Exit" button. The student might retrieve a program file by pressing the "Open" button and giving the file name. When the "Save As" button was pressed, the program in the edit window was saved on a file whose name was determined by the student. The "cc compiler" of the computer system was employed to execute the program when the "Run" button was pressed. If the program contained an error, it would be recorded by the lesson. For the study, students were asked to send programs which they modified or created. "Submit" button was used for this purpose.

Conceptual materials allowed students to see the internal operations of the computer and build a mental model. Manipulative activities and concrete models were provided in the conceptual materials. Two examples are shown in Figure 3.6 and Figure 3.7.

Figure 3.6 presents a concrete model which permitted students to perform a manipulative activity. Success in solving this problem required students to identify the problem, understand the task of each function, design a strategy to solve the problem, perform tasks with correct commands, and utilize the feedback message to correct errors. If students typed a syntactically correct command, the lesson performed the corresponding task. Students could examine the contents of variables or simulated output screen and see how the computer internally operated. If the command that students entered was not syntactically correct, a related feedback

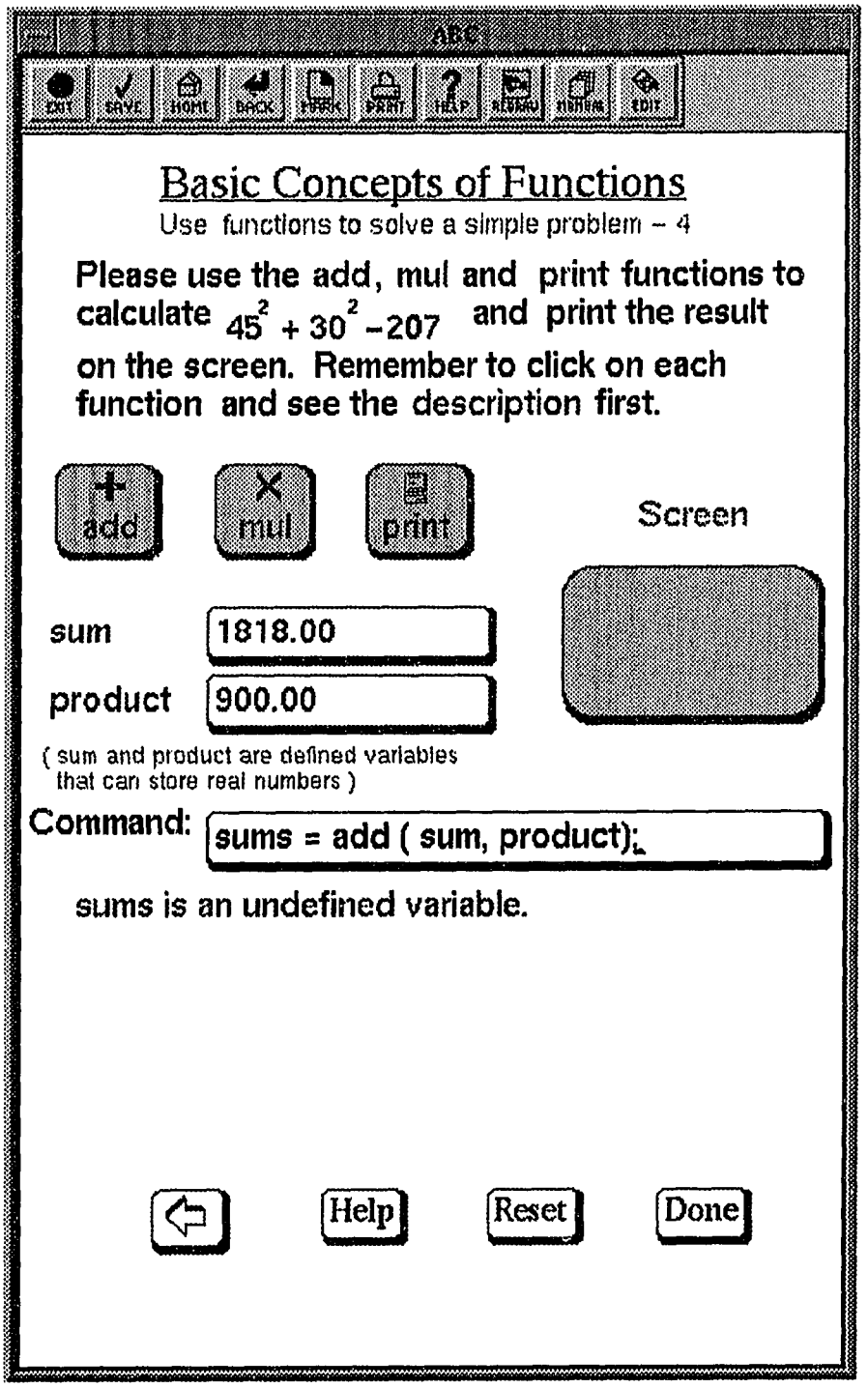


Figure 3.6: A manipulative activity of lesson function

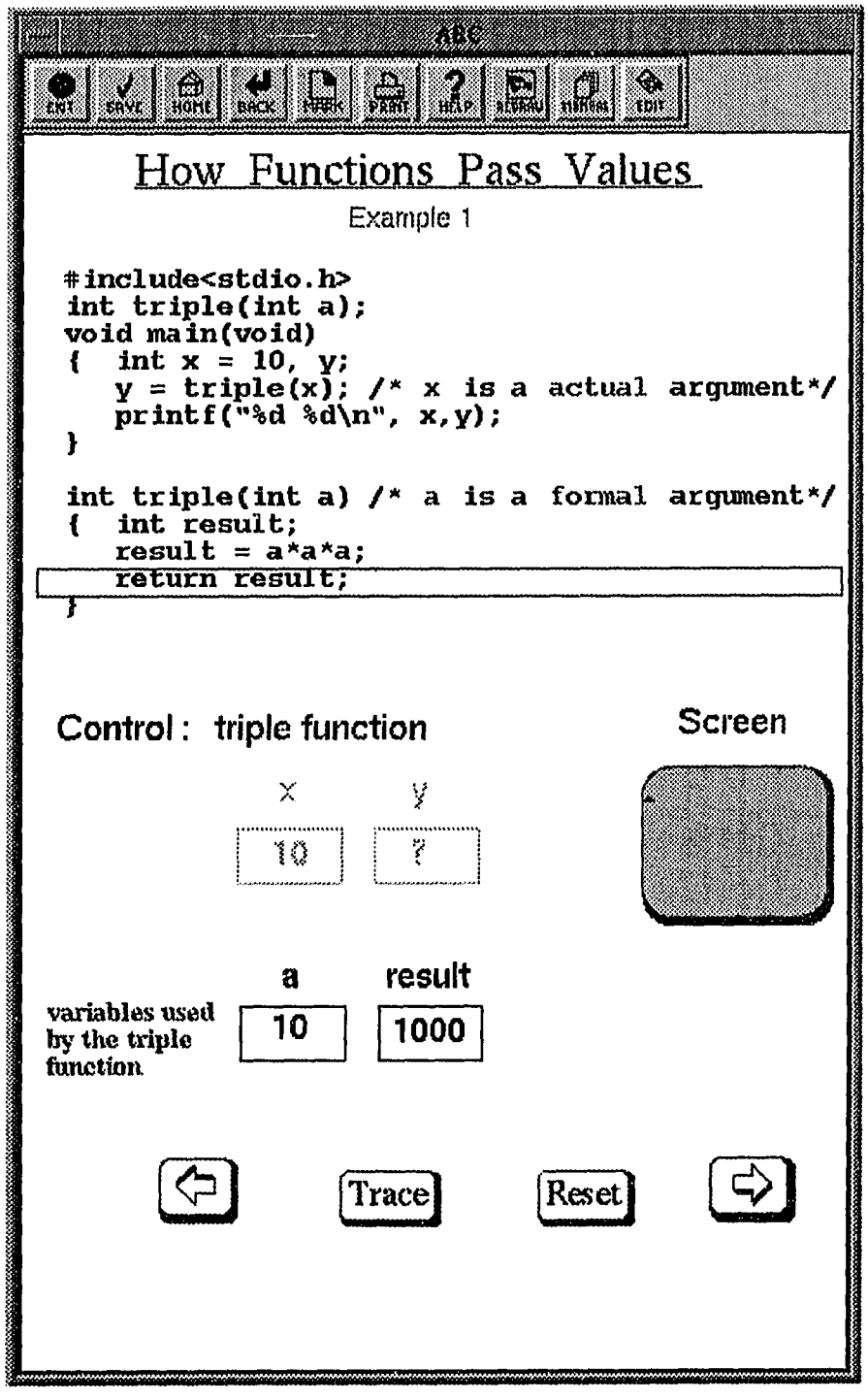


Figure 3.7: A concrete model of lesson function

message was provided. A great freedom in action was given to students. Students could press the "Reset" button and start the activity at any time, and they were able to try and test any of their own ideas.

Figure 3.7 shows a concrete model. This model consisted of several components: a program list, program control message, memory areas, and a imitative computer screen. Students could press the "Trace" button and see the execution of the program step by step. The internal computer operations were presented by text, graphics, and animation. The text information showed the entire program, the steps utilized in program execution, and the functions which controlled the program. The text information helped students build a clear concept of the program flow. The graphical materials displayed the memory contents of variables and the content of the simulated output screen. Viewing the change of variables and the screen provided a way to connect a program statement with the corresponding internal computer status. Furthermore, animation was employed to promote motivation in learning by showing the detailed processes of how a function passes values and influences the program. This concrete model attempted to help students understand difficult and abstract concepts in a concrete way.

Some exercises were given accompanied by the instructional materials. The exercises provided students the opportunity to utilize their existing knowledge to solve problems, confront insufficient knowledge, and re-evaluate current knowledge. Exercises included comprehending programs, identifying program errors, modifying and creating programs. When revising or writing a program was necessary in doing the exercises, the programming environment presented in Figure 3.5 was always given. The communication tool in the lesson granted students opportunities to ask ques-

tions during the learning process. Their questions were answered through the E-mail system.

Instruments

Five types of instruments were used to collect data pertinent to the study. The instruments of the study included a background questionnaire, a lesson function questionnaire, a lesson function feedback form, a pretest, and a posttest.

The background questionnaire was used to obtain information concerning number of prior computer courses, computer ownership, and a brief measure of students' attitudes toward taking the programming course. The estimated reliability of the attitude scale was 0.74 based on C'ronbach's alpha coefficient. The background questionnaire was completed by students prior to the experiment. A copy of it is shown in Appendix A.

The lesson function questionnaire was a open-ended questionnaire which was given to students at the end of the first section of learning *lesson function.* The purpose of this questionnaire was used to collect data regarding the problems that students encountered and obtain information related to the *lesson function.* A copy is presented in Appendix B.

The lesson function feedback form was completed by students after they finished the *lesson function.* It was used to gather information for improving the development of computer-based lessons and collect data concerning students' difficulties related to learning function concepts. Appendix C' shows the lesson function feedback form.

The pretest, Appendix D, was used to measure students' general programming knowledge prior to the experiment. The pretest was an open computer test which

contained three problems that involving the creation of working programs requiring basic concepts of any computer language. The estimated KR-21 (Kuder-Richardson Formula 21) reliability of the pretest was 0.73.

The posttest was conducted to evaluate students' achievement in C' function concepts after the experiment and analyze student difficulties related to use and creation of C' functions. The estimated KR-21 reliability of the posttest was 0.88. The posttest was given to students one week following the experiment. A copy of it can be found in Appendix E.

The posttest consisted of six major problems which measured students' ability to comprehend and write C' programs. The first problem on the posttest tested the understanding of how a function passes values. Students needed to realize that the data types, order, and numbers of actual arguments and formal arguments must be matched. The second problem required students to trace and write down the output of the C' program. Understanding of the manner in which C' programs pass control among functions was demanded. The third problem asked students to show the output of a C' program. Understanding the scope of local variables was necessary. The fourth problem required differentiating auto and static duration and determining the output of the program. The fifth problem asked students to rewrite a program by constructing one function to avoid repetitions of the same codes. Students needed to analyze the program and design a function that performs this specific task. The sixth problem required students to identify the problem, design an algorithm, and construct a program to solve the problem.

Using Bloom's taxonomy (1956), the first question of the posttest fits the application category in the cognitive domain. The remaining questions of the posttest can

be classified in analysis, synthesis or evaluation categories of the cognitive domain. All but the first question may be considered to test the transfer of problem solving methods.

Research Procedure

The research design for the study is a pretest posttest design consisting of two treatment groups. The subjects of the study were randomly assigned to either SEQUENCE1 or SEQUENCE2 experimental group. The number of subjects in these two groups were not the same because some students dropped the course during the experimental period. SEQUENCE1 group consisted of six subjects who received *lesson function* before traditional classroom instruction. SEQUENCE2 consisted of nine subjects who received lesson function after formal instruction. Both groups received the same lectures and completed the same assignments.

The class which subjects attended was a three credit hour course that met two times a week. Each class session was a two hour period. Before the experiment, fundamental knowledge of the computer, word processing packages, and the basic concepts of the C language were introduced to students. The course used the Borland Turbo C programming environment and the textbook "The Waite Group's C Programming Using Turbo C^{++}" (Lafore, 1993). Students were assigned chapter 5 in the text prior to the lecture. The lecture was combined with demonstration and practice. Each student utilized a 80486 PC for practice. Students learned the basic features of the C language including variables, data types, operators, input, output, conditional statements, and loop statements.

At the first week, the subjects were asked to complete the background ques-

tionnaire for collecting data on control variables. In order to help students become familiar with the computer environment, some materials were distributed to students (Appendix G), and students were asked to complete a *lesson vincent* outside the classroom environment. The *lesson vincent* was developed by the Computer Supported Learning Group at Iowa State University and is available on Project Vincent workstations. Students could access the *lesson vincent* at any time. This lesson provided both basic and advanced instruction on how to use the workstation, editing, filing, printing, and mailing. At the fourth week of the semester, the pretest was given to all subjects to evaluate their prior knowledge of programming.

The experiment started in the fifth week. In this week, the SEQUENCE1 group received *lesson function* in the reserved computer lab, and the SEQUENCE2 group was given formal instruction on the C function topic in the classroom. During the sixth week, the SEQUENCE1 group received the same lecture, and the SEQUENCE2 group worked on the *lesson function* in the reserved computer lab. Information sheets (Appendix H) guided students to start the lesson was given to students at the beginning of the first learning section.

During the experiment, the students' learning was observed, and some students' reactions were video-taped. At the end of the first learning section, students were asked to complete the lesson function questionnaire. After the experiment, the students were given the lesson function feedback form to complete. Finally, a week after the experiment, all students were given a posttest to measure their knowledge about the function concepts of the C language.

Data Analysis

The data collected from the instruments, observation, video-tape, and computer files regarding the learning of *lesson function* were used for analysis. Several files were created so that information could be easily stored, located, retrieved, and classified.

The quantitative data of the study was analyzed by using the SAS statistical package (SAS Institute, 1990) on the Project Vincent workstation at Iowa State University. Descriptive statistics such as means, standard deviations, frequencies were used to describe the general characteristics of subjects and the scale items of the lesson function feedback form. A t -test was conducted to test if differences exist between the two groups attitudes, number of prior computer courses, pretest, and posttest. A chi-square test was used to examine the independence of computer ownership. The Pearson product-moment correlation was applied to examine relationships among the variables. When testing the hypotheses, the multiple regression analyses described in the first chapter were used.

The investigator first browsed through the qualitative data of the study in order to get a overall sense of information and remove any redundant or unrelated data. After that, the data was reviewed and analyzed into categories which contained relevant materials. Both description and interpretation, with pertinent examples, were used to report the results.

CHAPTER 4. RESULTS

The results summarized in this chapter are based on data collected from research instruments (the background questionnaire, the lesson function questionnaire, the lesson function feedback form, the pretest, and the posttest), observations, videotapes, and the files obtained through the *lesson function*. Fifteen subjects participated in the study. All of them completed the background questionnaire, worked on the *lesson function*, and took the posttest. Of the fifteen students, fourteen took the pretest, and thirteen answered the lesson function questionnaire and the lesson function feedback form. The findings of this study are presented in four major sections: (1) analysis of subject background, (2) testing of the hypotheses, (3) students' reactions and difficulties and (4) information related to the *lesson function*.

Analysis of Subject Background

Analysis of control variables

The data collected from the background questionnaire (Appendix A) was used to test for differences between the two groups before the experiment. A *t*-test was conducted to determine if differences existed between the two groups in the students' attitudes and in the number of prior computer courses. A chi-square test was used to test for independence of computer ownership.

Table 4.1: Comparisons of continuous control variable means by treatment group

Variable	Treatment	N	Mean	SD	t Value	df	Prob>T
Students' Attitudes	SEQUENCE1	6	24.33	2.16	0.76	13	0.46
	SEQUENCE2	9	23.11	3.48			
Computer Courses	SEQUENCE1	6	1.00	0.89	-0.30	13	0.77
	SEQUENCE2	9	1.22	1.64			

Table 4.2: Distribution of computer ownership by treatment groups

Computer ownership	SEQUENCE1	SEQUENCE2	Total
Have computer	3	5	8
No computer	3	4	7
Total	6	9	15
Chi-square = 0.045	Significance = 0.83		

The results of both tests indicated that no significant differences existed between the two groups on the control variables of students' attitudes, number of prior computer courses, and computer ownership (Table 4.1 and Table 4.2).

Analysis of students' prior knowledge of programming

Before the experiment, students were given a pretest to measure their programming knowledge. As showed in Appendix D, the pretest contained three programming problems. It was an open computer test. Students could use any computer language to solve the problems, but all students chose C programming language. One stu-

dent in SEQUENCE2 group did not take the pretest. His score was predicated by the posttest, computer attitudes, number of prior computer courses, and computer ownership.

The pretest mean of the SEQUENCE2 group (6.32) was higher than the mean of the SEQUENCE1 group (5.33), but a t-value of -0.45, with a p-value of 0.66, revealed no significant difference between the two groups' mean ratings. In other words, random assignment of subjects had produced treatment groups which were not statistically different on prior knowledge of programming. Comparison of the pretest means of the two treatment groups is presented in Table 4.3.

Table 4.3: Comparison of pretest means of treatment groups

Treatment	N	Mean	SD	t Value	df	Prob>|T|
SEQUENCE1	6	5.33	4.55			
				-0.45	13	0.66
SEQUENCE2	9	6.32	3.86			
Total	15	5.93	4.02			

Testing of the Hypotheses

Findings from the posttest

The posttest shown in Appendix E contained six major problems. Students were not permitted to use a computer during the test. The SEQUENCE1 students had higher mean scores for all problems except the third problem of the posttest. Findings are reported in Appendix F. The posttest overall mean of SEQUENCE1 students was 30.33, and the overall mean of SEQUENCE2 students was 26.78. The

posttest mean of the SEQUENCE1 group was 3.55 units higher than the mean of the SEQUENCE2 group. However, a t-test showed that the posttest means of the two groups did not differ significantly (t-value = 0.71, df = 13, p-value = 0.49.) Comparison of the posttest means of the treatment groups is reported in Table 4.4.

Table 4.4: Comparison of posttest means of treatment groups

Treatment	N	Mean	SD	t Value	df	Prob>|T|
SEQUENCE1	6	30.33	10.13			
				0.71	13	0.49
SEQUENCE2	9	26.78	9.01			
Total	15	28.20	9.29			

The first problem of the posttest contained five subquestions. Students' answers revealed that they had trouble understanding the correspondence between the actual arguments used in calling a function and the argument list declared in a function prototype. Many students could not recognize the errors produced by mismatching data types and number of arguments. Furthermore, based on students' explanations, it was obvious that some students tried to correspond the arguments by variable name but not the data type. For example, in the first subquestion, one student explained that "the variable c is not declared in the prototype of function A. It should be a = A(a);." Other students made similar statements. However, four students, two from the SEQUENCE1 group and two from the SEQUENCE2 group, made no mistakes or only one mistake. All of these students had taken at least one computer course and had programming experience in BASIC, FORTRAN, Pascal or Logo.

All SEQUENCE1 students and six SEQUENCE2 students correctly answered

the second question of the posttest. This showed that most students were able to trace a C program consisting of many functions and successfully write down the program outputs. However, two students were confused by functions that call other functions. One student believed that the program tasks were executed in the order that functions were displayed, rather than in the order that functions were called.

One student from SEQUENCE1 and four students from SEQUENCE2 gave correct responses for the third problem. The other students did not completely understand the scope of local variables. Two or more functions can have local variables with the same names, but such identically named variables may be totally independent of each other and have their own values.

For the fourth question, two students from SEQUENCE1 and two students from SEQUENCE2 answered the question correctly. The other students had difficulty distinguishing the difference between auto and static variables and produced incorrect outputs of a program. Results from the second, third, and fourth questions indicated that students performed better in tracing a program flow that did not have data flow among functions, and they had trouble dealing with a program that had data flow among functions. In other words, argument passing and variable declaration are more abstract concepts for students to understand.

Solving the fifth problem required students to comprehend a C program and design a function to perform a specific task to avoid repetitions of the same code. Several students failed to conduct such a function because substantial reasoning skills were needed. From the students' programs, three significant errors were most commonly found. These included missing necessary arguments, passing unnecessary arguments and redefining variables. Figure 4.1, Figure 4.2 and Figure 4.3 shows three examples.

Figure 4.1 shows that a function call "out();" in the main function did not pass anything to "out" function. Since a necessary integer argument of "out" function was missed, the program could not perform the expected task. In addition, variables "i" and "j" were more appropriately defined in the "out" function rather than in the main function. The student who wrote this program had BASIC, FORTRAN and C programming experiences before this class. The errors that occurred might be the result of carelessness or misconceptions from preprogramming knowledge.

Figure 4.2 gives an example of passing unnecessary arguments. In this example, variable "i" and "j" were not required to be passed through arguments; instead, they could be declared inside the "star" function. Other errors could be found in the

```
#include<stdio.h>
void out(void);
void main(void)
{       int i,j,k;
        for (k = 1; k <= 3; k++)
           out();
}
void out(void)
{       for(i = 1; i <= k; i++)
    {       for ( j = 1; j <= k; j++)
              printf("*");
            printf("\n");
    }
}
```

Figure 4.1: An example of missing a necessary argument

example; for example, the data types of arguments had to be declared in the prototype and the function definition, variable "b" needed to be defined in main function, and the char return data type was unnecessary in star function. Further, using a "for loop" structure in the main function would have been more efficient.

```
#include<stdio.h>
char star (i, j, b);
void main(void)
{       b = 1;
        star (i, j, b);
        b = 2;
        star (i, j, b);
        b = 3;
        star (i, j, b);
}
char star (i, j, b)
{       for (i = 1; i <= b; i++)
    {       for (j = 1; j <= b; j++)
              printf("*");
            printf("\n");
    }
}
```

Figure 4.2: An example of passing unnecessary arguments

Figure 4.3 presents an example of redefining a variable. The variable "a" had been passed by an argument, yet it was redefined inside the "printstar" function. These three most significant errors revealed that students had difficulties understanding how functions pass values.

Missing a necessary argument and redefining a variable were errors found fre-

```
#include<stdio.h>
void printstar(int);
void main(void)
{       int i;
        for ( i = 1; i <= 3; i++)
           printstar (i);
}
void printstar(int a)
{       int i, j, a;
        for ( i = 1; i <= a; i ++)
    {       for (j = 1; j <=a; j++)
              printf("*");
            printf("\n");
    }
}
```

Figure 4.3: An example of redefining a variable

quently in students' answers to the sixth problem. Results also showed that several students had difficulties in decomposition. Some students tried to break a problem into subproblems that were similar to each other or did not execute all the tasks. Figure 4.4 shows an example. In this example, "positint" and "negint" were two similar functions that could be accomplished by a single function. Furthermore, functions constructed in the program could not perform all the tasks because the main function never called "positint" or "negint" functions. An additional function concerned with the sign of the integer was needed. The lack of appropriate connection between computer codes revealed that the student had serious problems in putting his code

into a logical sequence. Many students encountered this difficulty. This example also illustrated other errors, such as omitting the declaration of variable "x" in the main function and the redeclaration of variable "x" in the "positint" and "negint" functions.

```
#include<stdio.h>
void positint(int);
void negint(int);
void main(void)
{       printf("Please enter an integer:");
        scanf("%d", &x);
        if (x == 0)
        {       printf("\n Zero");
        }
}
void positint(int x)
{       int x;
        if (x > 0)
        {       printf("\n Positive Integer");
        }
}
void negint(int x)
{       int x;
        if ( x < 0 )
        {       printf("\n Negative Integer");
        }
}
```

Figure 4.4: An example of error in the design and plan

Analysis of hypotheses

Hypothesis 1: There is no significant contribution of the treatment variable to the prediction of posttest score variance among the subjects.

This hypothesis compared the effectiveness of a computer-based lesson used before versus after formal instruction in learning the function concepts of the C programming. The Pearson product-moment correlation was applied to test relationships among the variables. The results in Table 4.5 reveal that there were no significant relationships between any two variables.

A full model of a multiple regression analysis stated in Chapter 1 was conducted to predict the posttest results. The predictors were treatment, the pretest, students' computer attitudes, number of prior computer courses, computer ownership, and the

Table 4.5: Correlations among variables

Variable	1	2	3	4	5	6
1. Posttest	1.00					
2. Treatment	-0.19	1.00				
3. Pretest	0.46	0.13	1.00			
4. Attitudes	-0.16	-0.21	-0.33	1.00		
5. Courses	0.11	0.08	-0.43	0.15	1.00	
6. Ownership	-0.26	-0.05	-0.02	-0.01	0.01	1.00

interaction of treatment and the pretest. The R-square value of 0.36 and the p-value of 0.62 indicated that there were no predictors that significantly predicted the dependent variable (posttest). The result of the multiple regression is presented in Table 4.6.

Table 4.6: The result of the full model

Source	DF	Sum of Square	Mean Square	F	Pr > F
Model	6	437.41	72.90	0.76	0.62
Error	8	770.49	96.31		
Total	14	1207.90			

R-square = 0.36

Parameter	Estimate	T for H_0:parameter = 0	Pr> |T|
Intercept	30.15	1.14	0.29
Pretest	1.35	1.29	0.23
Treatment	-3.17	-0.34	0.74
Courses	-0.55	-0.24	0.82
Attitudes	-0.17	-0.17	0.87
Ownership	-4.68	-0.92	0.39
Interaction	-0.33	-0.24	0.81

Because the full model was not significant, the treatment predictor was not tested. Hypothesis 1 could not to be rejected at the 0.05 level. The results showed that whether the computer-based lesson was presented before or after the formal instruction made no difference in students' achievement as measured by the posttest.

Hypothesis 2: There is no significant portion of posttest score variance explained by pretest score variance.

This hypothesis examined whether students' prior knowledge in programming affected their learning of function concepts in C programming. The full model, the same as in testing Hypothesis 1, indicated that no significant differences were found (see Table 4.6). Therefore, Hypothesis 2 failed to be rejected at the 0.05 level. This indicated that students' prior knowledge in programming did not significantly affect their learning of function concepts in C programming.

Hypothesis 3: There is no significant interaction between the instructional sequence treatments and student prior knowledge as measured by the contribution of interaction to the prediction of posttest score variance.

This hypothesis tested whether there was any effect of interaction between instructional sequence and students' prior knowledge in programming on student achievement in learning the C function concepts. Similarly, the same full model used in testing Hypothesis 1 revealed that no significant differences were found (see Table 4.6). Therefore, Hypothesis 3 failed to be rejected at the 0.05 level. The results indicate that there was no combination effect of prior knowledge with instructional sequence (interaction) on learning C function concepts.

Students' Reactions and Difficulties

Students were asked to complete the lesson function feedback form (Appendix C) after they finished the *lesson function*. Question seven in this questionnaire asked students to indicate their feelings about learning the function concepts. A seven-point Likert-type scale was used to describe the degree of difficulty, from 1 = very easy to 7 = very hard. Descriptive statistics of these 10 sub-questions are presented in Table 4.7.

Table 4.7: Descriptive statistics of students' feelings

Question	N	Mean	SD	Min	Max
a. The syntax of the function	13	4.08	1.44	2	7
b. The return data type of a function	13	4.31	1.44	2	7
c. Deciding the argument list of a function	13	4.77	1.42	3	7
d. The calling function and called function	13	4.62	1.66	1	7
e. The flow of functions	13	4.77	1.64	2	7
f. The structure of C programs when using functions	13	3.92	2.05	1	7
g. The way that functions pass values	13	5.23	1.09	3	7
h. The global and local variables	13	5.15	1.91	1	7
i. The auto and static variables	12[a]	5.17	2.17	1	7
j. The strategies to solve a problem	13	5.38	1.33	3	7

[a]One subject did not answer the sub-question i.

The results showed that the strategies to solve a problem (mean = 5.38) and the way that functions pass values (mean = 5.23) were more difficult for students, whereas the structure of C programs when using functions (mean = 3.92) and the syntax of the function (mean = 4.08) were easier concepts to understand.

Question 4 in the lesson function feedback form measured the level of enjoying programming by using the C language. A seven-point Likert- type scale was used to describe the degree of enjoying, from 1 = not at all to 7 = very much. The mean score on this question was 3.46. Question 5 in the lesson function feedback form evaluated the degree of students' confidence in writing C programs that utilize functions. A seven-point Likert-type scale was used to describe the level of confidence from 1 = need a lot of help to 7 = feel confident. The mean score on this question was 2.54. Table 4.8 reports these findings. The results showed students did not enjoy C programming very much and did not have much confidence when writing C programs which use functions. Not surprisingly, it was difficult for students to be motivated and to maintain their confidence when they faced an enormous number of difficulties and abstract concepts.

Table 4.8: Descriptive statistics of enjoying programming and confidence in C programming

Question	N	Mean	SD	Min	Max
Do you enjoying the C programming?	13	3.46	1.13	2	5
How confident are you in writing C programs?	13	2.54	1.33	1	5

Most students convinced themselves that there are some advantages in using functions in programming. The advantages reported by them were:

1. breaking a program into small manageable parts,
2. organizing the aspects of a program,
3. avoiding duplicate codes or reusing the same codes in other programs,
4. making programs easier to comprehend, debug, test, and modify, and
5. permitting many different programmers to work on a program together.

When working on *lesson function*, students displayed different learning patterns. Some students immediately sought help upon encountering problems. Some students hesitated to ask for help even though they strongly needed it. A few students with more programming experience preferred to struggle to find solutions by themselves. Two students from SEQUENCE2 asked questions through the communication tool provided by the *lesson function*. Some students entirely completed an exercise before going on to the next section. In contrast, some students skipped exercises that they had trouble solving and advanced to other materials. One student did not work on any of the exercises. A few students went back to previous exercises when they developed an idea of how to solve the problem. Several students modified example programs and tested their own ideas. Two students took notes, especially copying down variable contents when tracing a program. One student often skipped plain text information and focused on examples and graphic information. It was found that SEQUENCE1 students spent more time than SEQUENCE2 students at the beginning. The instructor indicated that the SEQUENCE1 students seemed to pay

more attention during the formal instruction. Two students from SEQUENCE2 indicated that if they had choice they preferred to learn the *lesson function* before formal instruction because that would have provided more help to them.

When operating the manipulative models, some students were able to utilize feedback messages, examine the contents of variables, and solve problems quickly. Other students did not fully understand the problems, had difficulty designing a solution, and believed that the content of a variable is related to the meaning of its name. For example, some students indicated that the sum of two numbers should be assigned to the "sum" variable and could not be assigned to the "product" variable.

The examples presented were good resources for students to use in solving the problems. When students did the exercises, many of them went back to review the examples. Some students could quickly find the examples related to the exercises and continue their work, but some students blindly searched and copied the examples without actually understanding them.

Several students, especially those in the SEQUENCE1 group, reviewed the help questions and examples provided in a subsection of the *lesson function*. It appeared that the questions:

1. What's the return data type of this function?
2. What's the name you want to call this function?
3. How many arguments will be passed to this function, and what are their names and data types?
4. Are there any variables needed to be declared inside the function, and what are their names and data types?

5. What are the statements which can perform the specific task of the function?

might help them clarify their thinking and guide them in designing a function without worrying much about syntax. Unfortunately, from the questions that students asked, it appears that some students did not quite understand the data types, variables, and statements of C' language. That interfered with their learning about function concepts. Also, doing the exercises was made more difficult by unfamiliarity with the basic concepts of the C' programming language, such as input, output, assignment, conditional statements, and loop concepts.

When writing a program, none of the students wrote down their plans on paper. That means all students combined the design and the coding activities. They constructed solutions in their minds, directly translated solutions into statements and entered computer codes immediately.

Although students reported the syntax of the C' language was not a hard topic for them, all students made syntax errors, which resulted in programs that could not be executed successfully. Inability to find the cause of programming errors was a very frustrating part of learning to program. Some students continued to run a program with errors without changing any code. Some students just changed the first error that they could find and immediately ran a program again. Only a few students detected errors in detail before they reran programs. These findings revealed that most students debug programs by a trial-and-error approach.

An interesting finding was that some students had the ability to solve problems but had difficulty transferring their thoughts into a correct C' program. When doing an exercise, one student reported that he knows how to find the greatest common divisor of three positive integers, but he doesn't know how to map this knowledge

into C language. Another student indicated he is pretty good at finding the roots of a quadratic equation by hand but not at writing a program to solve it by the computer. These findings revealed that many students struggled with C language features.

A few students' codes contained some errors which were caused by confusing the features of another computer language with the C language. For example, a statement "y = a*x^2 + b*x + c", which is valid in a BASIC program but not for the C language, appeared in two students' programs. A student with Pascal experience was used to defining variables in the main function and attempted to use these variables in other functions. The variables defined in a Pascal main program are accessible in other functions, but variables defined in a C main function are invisible in other functions. Therefore, confusions occurred. A few students brought algebra knowledge to design a program; hence, statements like "x + y = z" occasionally appeared in their codes.

Some students had preprogramming knowledge, but they applied it incorrectly. Some exercises of the *lesson function* were designed to ask students about debugging a program containing errors and describing those errors. Figure 4.5 presents an example.

One student described the error and stated that "In the main function it needs to call the sum function before the printf command." He did not recognize the number of arguments mismatched between a function call and a called function, but made this explanation. This student had BASIC and FORTRAN programming experiences and might have learned that input, process, and output are three major tasks in a program. Because these three tasks usually were presented in a sequential way, the student thought it is reasonable to reach this conclusion.

```
#include<stdio.h>
int sum(int a, int b);
void main(void)
{       int a, b, c;
        scanf("%d %d %d", &a, &b, &c);
        printf("sum(%d %d %d) = %d \n", a, b, c, sum(a, b, c));
}
int sum (int a, int b)
{       return ( a+ b);
}
```

Figure 4.5: Identifying an error

Many syntactic errors found in students' programs were related to the semicolon. These included forgetting a necessary semicolon, adding an unnecessary semicolon, and replacing a semicolon with a comma. Other common errors were adding an unnecessary comma, using /n instead of \n, placing & before variables in printf statement, omitting & before variables in scanf statement, using printf function without giving format specifiers, using & instead of &&, using = instead of ==, declaring reserved words as variable names, including unnecessary header files, missing necessary header files, and forgetting necessary initialization. Some examples are shown in Table 4.9. These kinds of errors were found to be widespread.

Students generally did not have sufficient experience to determine which language structure was best for a problem. For example, students used several "if" statements instead of a " switch" statement, which is easier to read and more efficient. This type of problem also was found in choosing "for loop" or "while loop" structures.

Table 4.9: Examples of common errors

Error	Error Examples
1. Forgetting a necessary semicolon	x = a − b + c
2. Adding an unnecessary semicolon	void main(void);
3. Replacing semicolon with comma	for (i=1, i<=10, i−−)
4. Adding unnecessary comma	int sign (int, n)
5. Using /n instead of \n	printf("%d /n",answer);
6. Placing & before a variable in printf statement	printf("%d", &x);
7. Missing & before a variable in scanf statement	scanf("%d", a);
8. Using printf function without giving format specifiers	printf(i);
9. Using & instead of &&	if (a > 0 & a < 30) return a;
10. Using = instead of ==	if (n = 0) printf("Zero");
11. Declaring reserved words as variable names	int case;

12. forgetting necessary initialization	int sum(int n)
	{ int i, total;
	for (i=1; i<=n; i−−)
	total = total − i;
	return total;
	}

Two occasional errors involved redefining a function with a different argument list and declaring a variable name but using it as a function name. These errors suggested that some students might think the computer is smart enough to recognize the difference and interpret their codes. Several students reported that the concept of function prototypes was hard to understand, mostly because of the similarity of the headline of a function definition and the function prototype. A typical function prototype looks exactly like the headline of a function definition, with an appended semicolon. However, the C language allows an alternative for defining the function prototype. Thus, some confusion occurred. For example, "int area (int, int);" is a valid prototype, but "int area (int, int)" is not a valid function headline. Another problem was using variable names instead of data type in an argument list. For instance, students used statements like "int triangle (base, height);" as prototypes.

Choosing the return data type of a function and the argument list were two major problems of students. Students' errors concerning the return data type of a function included choosing a wrong data type, missing a return statement in the body of a function, returning a value with incompatible data type, and assigning a void function to a variable. Figure 4.6 shows an example. The function of this example attempted to calculate the area of a circle and return this value. The void return data type of this function was an incorrect choice, and this function returned a value with an incompatible data type.

Students' difficulties related to the argument list consisted of missing necessary arguments, passing unnecessary arguments, and choosing wrong data types for arguments. Another common error was to repeat defining variables inside a called function which had been passed by arguments. As described in a previous section,

```
void area(int r)
{ return (3.14159 * (r * r));
}
```

Figure 4.6: Example of an error concerning the return data type

three significant errors found in the posttest were: (1) missing necessary arguments, (2) passing unnecessary arguments, and (3) redefining variables. These errors arose frequently because at the beginning learning stage, students usually did not have accurate pictures about internal operations of the computer. Errors related to variables and argument passing were hard to eliminate until students had sufficient semantic knowledge.

Some students felt frustrated because they could not find a way to solve the problem. Inability to design a solution plan prevented several students from doing exercises or caused them to write programs with poor connections. As noted earlier, several students skipped some exercises when they encountered problems; one student did not even complete one exercise, and many students had trouble putting their codes in a logical order to achieve the problem task. This happened especially in solving exercises that required a substantial level of mathematic knowledge. Some serious difficulties for students were: (1) lack of problem domain knowledge, (2) shortage of syntactic and semantic knowledge, and (3) weakness in designing a solution. As a matter of fact, most students reported that writing a program was the most difficult activity to complete.

Information Related to the Lesson Function

Question 1, 2, 3 and 6 of the lesson function feedback form were concerned with the students' responses about *lesson function* used in the study. Findings are represented in Table 4.10. Other findings from the lesson function questionnaire and the lesson feedback form are reported below.

Table 4.10: Students' responses on feedback form

Question	N	Mean	SD	Min	Max
Did the lesson start at the right level?	13	4.92	0.95	4	6
Did the lesson provide appropriate examples?					
(a). Frequency	13	3.54	0.97	2	6
(b). Difficulty	12[a]	4.42	1.08	3	6
Did the lesson provide appropriate exercises?					
(a). Frequency	13	4.38	1.12	3	6
(b). Difficulty	12[a]	5.08	1.08	4	7
Did the lesson give you any help in learning function concepts?	13	4.15	1.40	2	7

[a]One subject didn't answer the question.

All students indicated that they had no trouble starting the lesson. Most students reported that the computer system did not crash or hang-up when they learned the *lesson function*. One student did have trouble because he accidentally hid the lesson under another window, but the problem was solved quickly. Most students stated that no section or activity was a waste of time. Two students said that the

early sections were too easy for them.

Many students indicated the pace was satisfactory for them; one student reported that it was too slow, and three stated that it was too fast. Features that students liked best about the *lesson function* were the following:

1. Many students indicated that the *lesson function* was well organized and user-friendly, with clear directions and instructions.

2. Some students reported that they could work at their own pace and look at the examples.

3. One student stated that the lesson provided a good number of examples, which gradually increased in difficulty.

4. A student said that the lesson provided visualization help to promote understanding of what is going on.

Features that students liked least about the *lesson function* were the following:

1. Some students indicated that more time was needed to complete the lesson.

2. Some students reported that some of the material concepts were hard to understand, especially for those without a programming background.

3. Some students indicated that exercises were difficult for them.

4. One student suggested that the programming environment used in the lesson was different from the Turbo C environment used in the class, and sometimes it caused confusion.

5. One student reported that more help information should be available to students in doing exercises.

Many students provided suggestions or comments for improving the lesson. These included:

1. Providing more examples related to exercises,
2. Cutting back on harder exercises and offering easier exercises,
3. Providing more detailed help information, especially in doing exercises,
4. Describing variables, data types, input, output, test and loop statements with short programs,
5. Making the programming environment consistent with the Turbo C programming environment, and
6. Permitting the examination of the variable values during debugging.

CHAPTER 5. SUMMARY AND DISCUSSION

This chapter provides an overview of the study. It is divided into six major sections: (1) summary, (2) limitations, (3) discussion, (4) issues related to teaching C programming language, (5) recommendations for future research, and (6) conclusion.

Summary

The purposes of the study were to investigate the effectiveness of presenting a computer-based lesson before versus after formal instruction and to examine the difficulties that students encountered in learning the function concepts of the C programming language.

The subjects used in the study were fifteen Industrial Education students enrolled during the fall semester of 1994 at Iowa State University. Subjects were randomly assigned to either the SEQUENCE1 or the SEQUENCE2 group.

To collect data pertinent to the study, the *lesson function* and instruments were developed. The *lesson function* was a computer-based lesson that was designed to help students learn the function concepts of the C language and that recorded students' programming errors, exercises and learning process information. Instruments of the study included the background questionnaire, the lesson function questionnaire, the lesson function feedback form, the pretest and the posttest.

Before the experiment, students were asked to complete the background questionnaire to provide data on control variables of the study, and the pretest was administered to measure their prior knowledge of programming. During the experiment, the SEQUENCE1 students received the *lesson function* before the formal lecture on the function topic of the C language, whereas the SEQUENCE2 students worked on the *lesson function* after a formal lecture that was identical to the lecture given to the SEQUENCE1 students. In addition, students were asked to complete the lesson function questionnaire, their learning processes were observed, and some students' reactions were video-taped. After the experiment, the posttest was administered to students to evaluate their learning of concepts of C language functions, and students completed the lesson function feedback form.

A *t*-test was applied to test whether the two groups differed with respect to attitudes, number of prior computer courses, pretest scores, and posttest scores. A chi-square test was used to examine the independence of computer ownership. When testing the hypotheses, multiple regression analyses were used. Students' reactions and programming errors were collected and divided into categories for analysis.

Based on the data analysis, the findings of the study were:

1. Students' performance, as measured by the posttest, was not affected by whether the computer-based lesson was presented before or after the formal lecture.

2. Students' prior knowledge in programming did not measurably affect the posttest scores.

3. There was no interaction between students' prior knowledge and instructional sequence.

4. Students struggled with syntactic problems that interfered with their learning the semantic knowledge and their problem solving ability. Many syntactic errors occurred because of the natural design of the C language.

5. All students combined the design and the coding activities. Students' difficulties in designing a solution plan and poor connection codes were found. In addition, most students used the trial-and-error approach during the debugging.

6. Choosing the wrong data type, missing a return statement in the body of the function, returning a value with incompatible data type, assigning a void function to a variable, missing necessary arguments, passing unnecessary arguments, choosing the wrong data types for arguments, redefining variables that had been passed by arguments, and designing similar functions were commonly found errors when students learned function concepts of the C language. In particular, students had difficult dealing with a program that involved data passing among functions.

Limitations

The findings were subject to three limitations of the study.

1. There were only fifteen participants in this study. This small sample size may not be representative of the population consisting of all individuals who could have participated.

2. The experimental time period was two weeks. Because the function topic involves complex concepts that require comprehensive understanding, exploring,

practicing and thinking, results of the study were restricted by the short time duration.

3. Only one researcher observed and interpreted students' learning in the study. Results may have been influenced by the researcher's knowledge, experience, and expectations.

Discussion

Discussion is organized into four sections: (1) placement of the *lesson function*, (2) students' prior knowledge of programming, (3) students' difficulties, and (4) effects of the *lesson function*.

Placement of the lesson function

Students who learned the *lesson function* before the formal lecture had scores that were slightly higher than, but not significantly different from, scores of students who received the reverse instructional sequence. However, some interesting findings were noted. The SEQUENCE1 students spent more time than the SEQUENCE2 students at the beginning of the learning *lesson function* . Consequently, less time remained for the SEQUENCE1 students to learn the later sections of the lesson. This might have affected their performance during the posttest. Further, the instructor indicated that the SEQUENCE1 students seemed to pay more attention during the formal lecture. Two students in the SEQUENCE1 group had very low pretest scores but very high posttest scores. This did not occur in the SEQUENCE2 group. In addition, two students in the SEQUENCE2 group indicated that if the *lesson function* had been introduced before they learned the C functions, it would have provided more

help to them. Because the study was limited by the small sample size and the short experimental time, increasing the sample size and the experimental time might have generated different results. Obviously, further research is needed.

Students' prior knowledge of programming

Pretest scores revealed that most students in the study were novice programmers, and only a few were experienced programmers. Although students' prior knowledge of programming did not produce a measurable effect on posttest scores, observation and files collected from the *lesson function* showed that some differences existed between experienced students and novice students.

When operating the manipulative models, experienced students were more skilled at understanding feedback messages, examining the contents of variables, and solving problems quickly. When debugging a program, experienced students examined the error codes before they ran the program. They completed more exercises and attempted to work more independently. In contrast, novice students had difficulties using the information given to solve the problem and interpreting error messages. They often ran erroneous programs without any modification or changed only one error during debugging. Surprisingly, some of them did not completely understand even the code they wrote. It was commonly found that novice students lacked the knowledge of basic features of the C language, such as input, output, conditional statement, and loop structures. This limited their understanding of the examples and their ability to do the exercises. This outcome supports Mayer's theory (1981) that for meaningful learning to occur, students must retain required knowledge to incorporate new information.

Experienced students, because of their previous programming experience, might benefit by the transferral of their previous programming knowledge to the learning of the C language. A problem they encountered was that some misconceptions they brought into the learning situation were related to their preprogramming experience. This finding supports the study of Bonar and Soloway (1985).

Students' difficulty

Many students were hard to motivate because learning programming is complex and difficult by nature. This observation is consistent with the report of Linn (1992). Furthermore, because the course was required for students, it might not meet their interests, needs, or ability and may have decreased their desire to learn programming. For meaningful learning to occur, the most important requirement is that students pay attention to the learning materials (Mayer, 1981). A high level of motivation is necessary to achieve meaningful learning. If students do not make enough effort to conquer the problems they face, there is a reduced likelihood for making progress.

Syntactic errors that prevent successful execution of programs and interfere with students' acquisition of semantic knowledge and problem-solving ability were commonly found. This finding supports Fay and Mayer's (1988) point that syntactic knowledge of programming is the basic requirement for successful programming. Many syntactic errors resulted from the design of the C language. The C language has been described as "scruffy" language that provides many powerful features but that is an ill-defined language with terse and unfriendly syntax (Green, 1990). For example, misuse of some operators confused students. The C language uses = as an assignment operator and == as an equal to relational operator. Students often

applied = instead of == as an equality comparison. Also, using && as logical and operator and using & as bitwise and operator and address symbol violates the experience of natural language. Therefore, students usually used & instead of && as logical and operator and frequently forgot to put & before a variable when the address of a variable was needed. Although these problems can be overcome by practice, they increased students' cognitive load during programming. Future computer language designers may need to be aware of and avoid these sources of confusion.

Students' programs often passed unnecessary arguments, omitted necessary arguments, redefined variables, and contained similar functions. They had particular difficulty dealing with data communication. These findings are consistent with reports of Carrasquel et al. (1989) and Fleury (1991b). Errors described above revealed students have inaccurate knowledge about how computers pass data among functions, allocate variables for use, and use data to perform tasks. Those internal misrepresentations resulted in wrong predictions, planning and interpretation of computer behavior. Lack of semantic knowledge obstructs effective programming (Fay & Mayer, 1988).

This study showed that students had difficulties developing a solution plan and had problems "putting the pieces of a program together" as described by Spohrer and Soloway (1986b). Lack of problem domain knowledge , the C language knowledge and experience of the design process itself might be the causes of these difficulties. Most students in the study used a trial-and-error approach to debug programs without understanding the program and spent most of their programming time in debugging; these findings are consistent with earlier studies (Allwood & Bjorhag, 1990; Nanja & Cook, 1987). Beginning students usually did not realize the importance of design

and of understanding the program and lacked the problem-solving ability related to these difficulties.

Students' difficulties discussed above raise questions for further studies:

1. Should computer programming be required for students other than majors in computer science or computer engineering?

2. If computer programming should be required for non-majors, can other courses, such as computer applications, problem-solving methods or logic reasoning, be taught before programming to help students master difficulties of programming?

In this information age, each student should be computer literate. To do so may require reviewing courses, making decision and changing the curriculum.

Effects of the lesson function

Students were found to be good at tracing a program that contains many functions without data passing. Previous research found that students frequently made the error of thinking that subprograms were executed in the order they appeared (Sleeman et al., 1986). In this study, only one student had this problem. It appears that utilizing both textual and visual means of representation to present knowledge decreases the chance of misunderstanding and enhances learning. One student indicated that the *lesson function* providing visualization allowed him to understand the internal operation of the computer, and this was hard to achieve by using a textbook.

Students particularly had trouble understanding the variable allocation and data passing. Although the *lesson function* provided concrete models to help them deal with this problem, because of time limitations, some students had no chance to

complete those materials or looked at those examples only once, without detailed thinking. Results showed that two students who repeatedly operated those models and took notes had high scores in the posttest problems that required these concepts.

The *lesson function* was limited by the fact that the programming environment provided was different from the Turbo C programming environment used in the classroom. This difference caused some confusion for students. Moreover, the concrete models provided in the *lesson function* were predefined examples. This restricted the flexibility of the instruction. If students could write any programs, trace their own programs step by step, and examine the contents of variables, they should be able to build more concrete and accurate mental models of the computer language. However, the *lesson function* provided an environment that allowed students to be involved and to explore programming activity in a way that promoted students' programming abilities.

Issues Related to Teaching C Programming Language

The C language is a complex language for beginners; therefore, some educators view this as a pedagogical disadvantage. Should a more simple computer language instead of the C language be taught in the first course of programming? This question remains for future researchers to investigate. However, whether the C language is taught at introductory courses or advanced courses, emphasis on instruction will provide help and reduce confusion for students.

Instruction may illustrate some common syntax confusion to help students avoid those problems. Using #define directive can modify the syntax. For instance, "#define AND &&" will substitute AND with && through the entire program. Therefore,

It allows students to use AND as a logical and operator and avoid the confusion caused by &&.

Since students writing the first C' program must involve the main function, giving them an overview of the C' functions before introducing the first C' program may be helpful. Instruction may explain that a C' program consists of functions. Functions can be divided into library and user-defined functions. Library functions are built in the C' language to help users input, process, and output data. User-defined functions are designed by users to accomplish specific tasks. The main function is a special user-defined function that is the first function executed by a C' program and must exist in a C' program. Students usually can accept these concepts without many questions. Later, when introducing library or user-defined functions in more detailed, the instructor can refer back to the function concepts. This approach gives students a whole picture first and helps them fill in the details later.

Simplicity should be kept in mind when teaching beginning students. Examples, which are good resources for students, are best used to introduce one new concept at a time so that students are not confused. For instance, when discussing user-defined functions, the first example should only contain functions that do not have any data communication with other functions. After that, the one way data communication, passing data from a calling function to a called function or returning data from a called function to a calling function, can be presented. Following that, examples can demonstrate the two ways data communication between a calling function and a called function.

The similarity of the function prototype and the headline of a function definition caused some confusion. The first example of a user function given to students needs

not to contain the prototypes. After students experience some examples and exercises about functions, the concept of the prototypes can be introduced. At this point, instruction should differentiate these two approaches of writing programs (bottom-up and top-down) and discuss their advantages and disadvantages.

The common errors in students' programs suggested that students do not have clear internal representations of the C language's semantics, structures, characteristics and rules. Based on the computer environment, building a visible and manipulative "notional machine" with examples and help information is believed to be a solution. The visible approach allows students to view a simulation of internal operations of the machine and hopefully enhance their understanding of how computers operate. The manipulative approach permits students to explore and practice programming activity, thereby promoting their problem-solving ability. Examples and help information will guide students in effective ways and eliminate their frustrations.

For example, the notional machine can be presented by an input area, an output area, a program area, an operation area, a message area, a "Check" button, a "Trace" button, and a "Help" button. The input area is used to key in data if a program contained an input statement. The program area presents predefined examples or user-designed programs. Commands supported in the machine can be introduced by the predefined examples. The "Check" button can direct the machine to check the syntax of the program. If syntax errors exist, related error messages will be provided in the message area. If the program is syntactically correct, a message that guides users to use the "Trace" button to trace the program is displayed in the message area. During the trace, (1) the operation area shows variable contents and demonstrates the action of the machine, (2) the message area explains the machine behavior, and

(3) the output area presents output results. When the "Help" button is pressed, users can access information of supported commands and related examples. However, the difficulty of developing such an environment is expected to be considerable; therefore, team work and a long term project may be necessary.

Specification, design, coding and debugging those cognitive activities required in programming should be introduced to students. These concepts provide students with a whole picture of how to develop a program. Flow charts, pseudocodes or other applicable tools and the concepts of top-down and bottom-up designs should be provided to students, with examples to follow and exercises to practice. Instruction should encourage students to write down their design plans, especially for complex problems. In addition, students should be taught how to interpret error messages, analyze output information, use comments and functions, simulate a program execution, and utilize available debugging tools to narrow the range of error codes and locate and correct errors. Once students reach a sophisticated level, they can apply knowledge and experience on their own and solve problems quickly.

It appears that given a limited amount of available time there is too much to teach. Designing effective instruction always challenges educators. More researchers are encouraged to contribute their effort in this area.

Recommendations for Future Research

Based on the experience gained from the study, the following recommendations are made for future research:

1. Replication of this study should use an enlarged sample to obtain more representative findings. Furthermore, a control group that does not receive a

computer-based lesson is better included in the study to quantitatively examine the effect of the computer-based lesson if enough subjects are available.

2. Future research should extend the experimental time, so that students have sufficient time to learn the computer-based lesson and be involved in substantial thinking and practicing.

3. A team approach that will make more detailed observations, produce more exhaustive findings, and make more contributions than a single researcher can make is recommended.

4. Interviews of students and use of think-aloud strategies may provide additional insights into students' thinking and allow for collection of more detailed information regarding their approaches to solving problems, which can be helpful in identifying students' learning difficulties.

5. The method of automatic recording of students' programs is useful in collecting data on programming behavior that has been difficult to observe. Comparing different program versions of a problem can reveal students' errors and their debugging processes. However, data analysis is a time-consuming process that needs to be considered by future researchers.

6. Investigating students' preferences and the reasons for their preferences regarding the sequencing of the computer-based lesson and the formal instruction may provide information about students' needs, and relationships among students' prior knowledge, preferences, and achievement can be also examined.

7. The function is only one of the important topics in the C language. Arrays and pointers that are powerful features of the C language and often cause confusion are topics that may require special investigations.

Conclusion

The computer is a complex device. Learning to program a computer is not a natural human behavior; it requires effort and various kinds of knowledge and experience. Many difficulties can be expected. Designing instruction to teach students how to program is even harder than programming itself. It requires not only knowledge of programming but also knowledge of students' learning and thinking. This study reveals students' learning difficulties and provides applicable suggestions for developing programming instruction. It is hoped that the study will encourage other researchers conducting further investigations in this area and that it has contributed to improve computer education.

BIBLIOGRAPHY

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. IEEE Transactions on Software Engineering, 11(11), 1351-1360.

Alexander, L., Frankiewicz, R., & Williams, R. E. (1979). Facilitation of learning and retention of oral instruction using advance and post organizers. Journal of Educational Psychology, 71(5), 701-707.

Allwood, C. M., & Bjorhag, C. G. (1990). Novices' debugging when programming in Pascal. International Journal of Man-Machine Studies, 33,707-724.

Anderson, J. R., & Reiser, B. J. (1985). The LISP tutor. Byte, 10,159-175.

Ausubel, D. P. (1960). The use of advance organizers in the learning and retention of meaningful verbal material. Journal of Educational Psychology, 51(5), 267-272.

Ausubel, D. P., & Fitzgerald, D. (1961). The role of discriminability in meaningful verbal learning and retention. Journal of Educational Psychology, 52(5), 266-274.

Ausubel, D. P., & Youssef, M. (1963). The role of discriminability in meaningful parallel learning. Journal of Educational Psychology, 54, 331-336.

Bailie, F. K. (1991). Improving the modularization ability of novice programmers. SIGCSE Bulletin, 23(1), 277-282.

Barnes, B. R., & Clawson, E. U. (1975). Do advance organizers facilitate learning? Recommendations for further research based on an analysis of 32 studies. Review of Educational Research, 45(4), 637-659.

Bloom, B. S. (Ed.). (1956). Taxonomy of educational objectives: The classification of educational goals, handbook 1: Cognitive domain. New York: David McKay Company.

Bonar, J., & Cunningham, R. (1988). Bridge: An intelligent tutor for thinking about programming. In J. Self (Ed.), Artificial intelligence and human learning: Intelligent computer-aided instruction (pp.391-409). New York, NY: Chapman and Hall.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. Human-Computer Interaction, 1(2), 133-161.

Boysen, P. (1992). A guide to ABC for programmers. Unpublished manuscript, Iowa State University, Computation Center, Ames.

Boysen, P. (1993). A reference to ABC types. Unpublished manuscript, Iowa State University, Computation Center, Ames.

Boysen, P. (1994). ABC - An object-oriented instructional system [ABC documentation on Mosaic]. Ames, IA: Iowa State University.

Brant, G., Hooper, E., & Sugrue, B. (1991). Which comes first the simulation or the lecture?. Journal of Educational Computing Research, 7(4), 469-481.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9(6), 737-751.

Brooks, R. (1983). Toward a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18, 543-554.

Brooks, R. (1990). Categories of programming knowledge and their application. International Journal of Man-Machine Studies, 33(3), 241-246.

Brusilovsky, P. L. (1993). Towards an intelligent environment for learning introductory programming. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 114-124). Berlin: Springer-Verlag.

Callihan, H. D., & Bell, F. H. (1978). The effect of specially constructed advance organizers and post-organizers on mathematics learning. International

Journal of Mathematical Education in Science and Technology, 9(4), 433-450

Carnes, E. R., Lindbeck, J. S., & Griffin, C. F. (1987). Effects of group size and advance organizers on learning parameters when using microcomputer tutorials in kinematics. Journal of Research in Science Teaching, 24(9), 781-789.

Carrasquel, J., Roberts, J., & Pane, J. (1989). The design tree: A visual approach to top-down design and data flow. SIGCSE Bulletin, 21(1), 17-21.

Confrey, J. (1990). A review of the research on student conceptions in mathematics, science, and programming. In C. B. Cazden (Ed.), Review of research in education (pp. 3-56). Washington, DC: American Educational Research Association.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. Journal of Educational Computing Research, 1(3), 253-274.

Dembo, M . H. (1994). Applying educational psychology (5th ed.). New York: Longman.

Doyle, W. H. (1986). Using an advance organizer to establish a subsuming function concept for facilitating achievement in remedial college mathematics. American Educational Research Journal, 23(3), 507-516.

du Boulay, B. (1986). Some difficulties of learning to program. Journal of Educational Computing Research, 2(1), 57-73.

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices. International Journal of Man-Machine Studies, 14, 237-249.

Eylon, B. -S., & Linn, M. C. (1988). Learning and instruction: An examination of four research perspectives in science education. Review of Educational Research, 58(3), 251-301.

Fay, A. L., & Mayer, R. E. (1988). Learning logo: A cognitive analysis. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 55-74). Hillsdale, NJ: Lawrence Erlsaum Associates.

Ferguson, D. L., Henderson, P. B. (1987). The design of algorithms. Machine-Mediated Learning, 2(1 & 2), 67-82.

Fleury, A. E. (1991a). Parameter passing: the rules the students construct. SIGCSE Bulletin, 23(1), 283-286.

Fleury, A. E. (1991b). The content knowledge, programming skills, and beliefs concerning parameter passing of college students enrolled in a Pascal programming course. Unpublished doctoral dissertation, University of Wisconsin, Madison.

Green, T. R. G. (1990). The nature of programming. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), Psychology of programming (pp. 21-44). London: Academic Press.

Gugerty, L., & Olson, C. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 13-27). Norwood, NJ: Ablex Publishing Corporation.

Hoc, J. M. (1977). Role of mental representations in learning a programming language. International Journal of Man-Machine Studies, 9, 87-105.

Hoc, J. M., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), Psychology of programming (pp. 139-156). London: Academic Press.

Hooper, E. J., & Thomas, R. A. (1990). Investigating the effects of a manipulative model of computer memory operations on the learning of programming. Journal of Research on Computing in Education, 22(4), 442-456.

Husic, F. T., Linn, M. C., & Sloane, K. D. (1989). Adapting instruction to the cognitive demands of learning to program. Journal of Educational Psychology, 81(4), 570-583.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquisition (pp. 255-283). Hillsdale, NJ: Lawrence Erlbaum Associates.

Johnson, W. L. (1990). Understanding and debugging novice programs. Artificial Intelligence, 42,51-97.

Jonassen, D. H. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm?. Educational Technology, Research and Development, 39(3), 5-14.

Joni, S. -N. A., & Soloway, E. (1986). But my program runs! Discourse rules for novice programmers. Journal of Educational Computing Research, 2(1), 95-125.

Kassab, V. (1989). Technical C programming. Englewood Cliffs, NJ: Prentice Hall.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in lisp. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 198-212). Norwood, NJ: Ablex Publishing Corporation.

Krahn, C. G., & Blanchaer, M. C. (1986). Using an advance organizer to improve knowledge application by medical students in computer-based clinical simulations. Journal of Computer-Based Instruction, 13(3), 71-74.

Kulik, C. -L. C., & Kulik, J. A. (1991). Effectiveness of computer-based instruction: An updated analysis. Computers in Human Behavior, 7(1-2), 75-94.

Lafore, R. (1993). The Waite group's C programming using Turbo C++ (2nd ed.). Indianapolis, IN: SAMS Publishing.

Lewis, W. E. (1980). Problem-solving principles for programmers: Applied logic, psychology, and grit. Rochelle Park, NJ: Hayden Book Company.

Lieberman, H. (1986). An example based environment for beginning programmers. Instructional Science, 14(3-4), 277-292.

Linn, M. C. (1992). How can hypermedia tools help teach programming?. Learning and Instruction, 2(2), 119-139.

Linn, M. C., & Clancy, M. J. (1992). Can experts' explanations help students develop program design skills?. International Journal of Man-Machine Studies, 36, 511-551.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Intruction , access, and ability. Educational Psychologist, 20(4), 191-206.

Luiten, J., Ames, W., & Ackerson, G. (1980). A meta-analysis of the effects of advance organizers on learning and retention. American Educational Research Journal, 17(2), 211-218.

Mayer, R. E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, 67(6), 725-734.

Mayer, R. E. (1976). Some conditions of meaningful learning for computer programming: Advance organizers and subject control of frame order. Journal of Educational Psychology, 68(2), 143-150.

Mayer, R. E. (1979a). Can advance organizers influence meaningful learning?. Review of Educational Research, 49(2), 371-383.

Mayer, R. E. (1979b). Twenty years of research on advance organizers: assimilation theory is still the best predictor of results. Instructional Science, 8(2), 133-167.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. Computer surveys, 13(1), 121-141.

Mayer, R. E. (1988). Introduction to research on teaching and learning computer programming. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple research perspectives (pp. 1-12). Hillsdale, NJ: Lawrence Erlsaum Associates.

Mayer, R. E. (1989). Models for understanding. Review of Educational Research, 59(1), 43-64.

Mayer, R. E., & Bromage B. K. (1980). Different recall protocols for technical texts due to advance organizers. Journal of Educational Psychology, 72(2), 209-225.

Morton, L., & Norgaard, N. (1993). A survey of programming languages in CS programs. ACM SIGCSE Bulletin, 25(2), 9-11,18.

Nanja, M., & Cook, C. R. (1987). An analysis of the on-line debugging process. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical studies of programmers: Second workshop (pp. 172-184). Norwood, NJ: Ablex Publishing Corporation.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. Journal of Educational Computing Research, 2(1), 25-36.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. New Ideas in Psychology, 2(2), 137-168.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), Psychology of programming (pp. 45-62). London: Academic Press.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, K., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Perry, G. (1992). C++ programming 101. Carmel, IN: SAMS Publishing.

Peterson, J. C., Lovett, C. J., Thomas, H. L., & Bright, G. W. (1973). The effect of organizers and knowledge of behavioral objectives on learning a mathematical concept. Journal for Research in Mathematics Education, 4(2), 76-84.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. Canadian Journal of Psychology, 39(2), 240-272.

Polya, G. (1957) How to solve it: A new aspect of mathematical method (2nd ed.). Princeton, NJ: Princeton University Press.

Putnam, R. T., Sleeman, D., Baxter, J. A., & Kuspa, L. K. (1986). A summary of misconceptions of high school Basic programmers. Journal of Educational Computing Research, 2(4), 459-472.

Ramadhan, H., & du Boulay, B. (1993). Programming environments for novices. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 125-134). Berlin: Springer-Verlag.

Ratcliff, B., & Siddiqi, J. I. A. (1985). An empirical investigation into problem decomposition strategies used in program design. International Journal of Man-Machine Studies, 22(1), 77-90.

Righi, C. (1991). Using advance organizers to teach BASIC programming to primary-grade children. Educational Technology Research and Development, 39(4), 79-90.

Rist, R. S. (1986). Plans in programming: Definition, demonstration, and development. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 28-47). Norwood, NJ: Ablex Publishing Corporation.

Roberts, E. S. (1993). Using C in CS1: Evaluating the Stanford experience. ACM SIGCSE Bulletin, 25(1), 117-121.

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J. M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), Psychology of programming (pp. 157-174). London: Academic Press.

Rogalski, J., & Samurçay, R. (1993). Task analysis and cognitive model as a framework to analyse environments for learning programming. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 6-19). Berlin: Springer-Verlag.

Rudd, A. (1994). Mastering C. New York: John Wiley & Sons.

SAS Institute (1990). SAS/STAT user's guide, version 6 (Vols. 1-2). Cary, NC: SAS Institute.

Segal, J., & Ahmad, K. (1993). The role of examples in the teaching of programming languages. Journal of Educational Computing Research, 9(1), 115-129.

Shneiderman, B., & Mayer, R. E. (1979). Syntactic/semantic interactions in programmer behabior: A model and experimental result. International Journal of Computer and Information Sciences, 8(3), 219-238.

Shuell, T. J. (1986). Cognitive conceptions of learning. Review of Educational Research, 56(4), 411-436.

Shuell, T. J. (1992). Designing instructional computing systems for meaningful learning. In M. Jones & P. H. Winne (Eds.), Adaptive learning environments: Foundations and frontiers (pp. 19-54). Berlin: Springer-Verlag.

Simplicio-Filho, F. C. (1993). A distributed model of cognitive behaviour in specification understanding. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 80-93). Berlin: Springer-Verlag.

Sleeman, D, Putnam, R. T., Baxter, J, & Kuspa, L. (1986). PASCAL and high school students: A study of errors. Journal of Educational Computing Research, 2(1), 5-23.

Sloane, K. D., & Linn, M. C. (1988). Instructional conditions in Pascal programming classes. In R. E. Mayer (Ed.), Teaching and learning computer programming: Multiple Research Perspectives (pp. 207-235). Hillsdale, NJ: Lawrence Erlsaum Associates.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. Communications of the ACM, 26(11), 853-860.

Spohrer, J. C. (1992). MARCEL: Simulating the novice programmer Norwood, NJ: Ablex Publishing Corporation.

Spohrer, J. C., & Soloway, E. (1986a). Analyzing the high frequency bugs in novice programs. In E. Soloway & S. Iyengar (Eds.), Empirical studies of programmers (pp. 230-251). Norwood, NJ: Ablex Publishing Corporation.

Spohrer, J. C., & Soloway, E. (1986b). Novice mistakes: Are the folk wisdoms correct?. Communications of the ACM, 29(7), 624-632.

Stone, C. L. (1983). A meta-analysis of advance organizer studies. Journal of Experimental Education, 51(4), 194-199.

Sumiga, J. (1993). Programming and design. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming (pp. 59-70). Berlin: Springer-Verlag.

Upah, S., & Thomas, R. A. (1993). An investigation of manipulative models on the learning of programming loops. Journal of Educational Computing Research, 9(3), 397-412.

van der Veer, G. C. (1993). Mental representations of computer languages - A lesson from practice. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), Cognitive models and intelligent environments for learning programming

(pp. 20-33). Berlin: Springer-Verlag.

Youngs, E. A. (1974). Human errors in programming. International Journal of Man-Machine Studies, 6,361-376.

ACKNOWLEDGEMENTS

I would especially like to express my sincere appreciation to Dr. William Miller, my major professor, for his patience, inspiration and guidance throughout my graduate study and research. I would also like to thank the other members of my committee: Dr. Rex Thomas for his insightful and invaluable comments, Dr. Robert Strahan, Dr. Roger Smith and Dr. Ronald Meier for their assistance, and Dr. Pete Boysen for his ABC' system and support.

In addition, I would like to thank the students who participated in the experiment for their contribution to the study. Gratitude is extended to professor Young-Ho Tsai for his remote inspiration and support. Appreciation is also expressed to all my wonderful friends and my sister for their encouragement and support during the frustrating time.

Special thanks are expressed to my mother and parents-in-low for their love and financial support. Most of all, I would like to thank my dear husband, Hsin-Feng, for his faith in my abilities and encouragement during the completion of my study. Great appreciation is also expressed to my lovely children, Wei-Hsuan and Yi-Ning, for the happiness and hope that they have brought to my life.

APPENDIX A. BACKGROUND QUESTIONNAIRE

Background Questionnaire

Note: All information provided on this questionnaire will be kept in strict confidence and will have no bearing in determining your course grade.

Social Security No:_____
Project Vincent Username:_____
Major:_____
Year in College:_____
Age:_____ Sex:_____

1. Please list any high school computer science courses you have taken.

   Course Name

   (1) _____
   (2) _____
   (3) _____
   (4) _____

2. Please list any college computer science courses you have taken.

   Course Name

   (1) _____
   (2) _____
   (3) _____
   (4) _____

3. Please check all the computer languages that you have used.

(1) _____ BASIC
(2) _____ Pascal
(3) _____ FORTRAN
(4) _____ COBOL
(5) _____ PL/1
(6) _____ LOGO
(5) _____ PL/1
(6) _____ LOGO
(7) _____ C
(8) _____ assembly
(9) _____ LISP
(10)_____ PROLOG
(11)_____ Others (Specify:_____)

4. Do you have a computer available at home for your use?

(1) _____ Yes
(2) _____ No

5. Please circle a number to indicate your feelings.

1 = Strongly Disagree
2 = Disagree
3 = Neutral
4 = Agree
5 = Strongly Agree

		SD	D	N	A	SA
a.	I am interested in using a computer to solve problems.	1	2	3	4	5
b.	I think using a computer will not be very hard for me.	1	2	3	4	5
c.	I have confidence that I will do well in this course.	1	2	3	4	5
d.	I expect to receive a good grade in this course.	1	2	3	4	5
e.	This course will improve my computer skills.	1	2	3	4	5
f.	This course will increase my future job opportunities.	1	2	3	4	5

APPENDIX B. LESSON FUNCTION QUESTIONNAIRE

Lesson Function Questionnaire

DIRECTIONS: Because we desire to continuously improve the quality of this instruction, we ask you to complete the following questions. Please be frank and honest and provide as much detail as will help improve the product. Thank you!

I. USING THE COMPUTER SYSTEM

1. Did you have any trouble starting the lesson? If so, what problems? How did you resolve the problem?

---

2. Did the system "crash" or "hang-up" while you were using it? If so, how did you get started again?

---

3. Do you have any suggestions for improving the directions for use of the computer? If so, what are they?

II. THE LESSON

4. Please identify or describe the one section or activity you have completed that was the most difficult to complete. Why do you think it was difficult? What did you finally do to complete it?

Most difficult

Why and how completed?

---

5. Were there any sections or activities which you felt were a waste of time? If so, how might they be improved?

---

6. How about the pace of the instruction? Is it too fast or too slow? How could it be improved?

---

7. Have you read the material in the text book covering the same concepts? If so, was it helpful to read it before doing the computer lesson?

Thanks again for your suggestions and comments!

APPENDIX C. LESSON FUNCTION FEEDBACK FORM

Lesson function feedback form

Please circle the appropriate responses or provide information to each question. Thank you!

1. Did the "Lesson function" start at the right level?

```
          1     2     3     4     5     6     7
          |-----|-----|-----|-----|-----|-----|
Too elementary        Appropriate       Too advanced
```

2. Did the lesson provide appropriate examples?

```
          1     2     3     4     5     6     7
          |-----|-----|-----|-----|-----|-----|
Too few             Fine            Too many
```

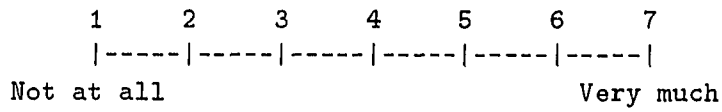
```
          1     2     3     4     5     6     7
          |-----|-----|-----|-----|-----|-----|
Too easy            Fine            Too hard
```

3. Did the lesson provide appropriate exercises?

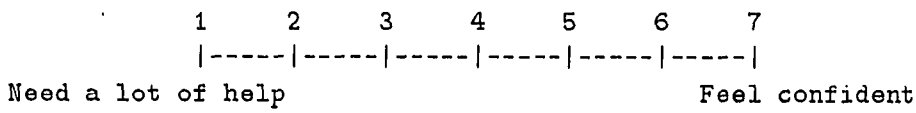
```
          1     2     3     4     5     6     7
          |-----|-----|-----|-----|-----|-----|
Too few             Fine            Too many
```

```
          1     2     3     4     5     6     7
          |-----|-----|-----|-----|-----|-----|
Too easy            Fine            Too hard
```

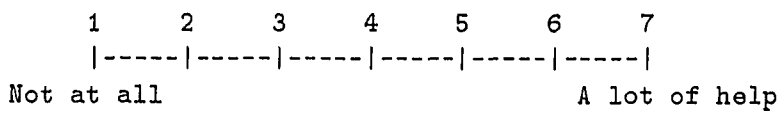
4. Do you enjoy programming by using the C language?



5. How confident are you in writing C programs that utilize functions?



6. Did the lesson give you any help in learning function concepts?



7. Please indicate your feeling about learning the following function concepts:

	Very easy 1 2 3 4 5 6 7 Very hard
a. The syntax of the function	|--|--|--|--|--|--|--|
b. The return data type of a function	|--|--|--|--|--|--|--|
c. Deciding the argument list of a function	|--|--|--|--|--|--|--|
d. The calling function and called function	|--|--|--|--|--|--|--|
e. The flow of functions	|--|--|--|--|--|--|--|
f. The structure of C programs when using functions	|--|--|--|--|--|--|--|
g. The way that functions pass values	|--|--|--|--|--|--|--|
h. The global and local variables	|--|--|--|--|--|--|--|
i. The auto and static variables	|--|--|--|--|--|--|--|
j. The strategies to solve a problem	|--|--|--|--|--|--|--|

8. What did you like best about the lesson?

---

9. What did you like least about the lesson?

---

10. What are your beliefs about the advantages of using functions?

---

11. Please provide any comments or suggestions to improve the lesson.

APPENDIX D. PRETEST

IEDT 216 TEST 1 Fall 1994

Student Name:_____ SS#_____

You can use any computer language to solve the problems 1 - 3. If you don't use the C language, please specify the language you use. You may use a computer if you wish.

1. [10 points] Write a program that gets three integers from the keyboard, compares the integers and outputs these integers from greatest to smallest.

2. [10 points] Write a program that computes the sum of scores that are entered from the keyboard until any negative number is encountered.

3. [10 points] Suppose that a person should pay income tax according to the following tax table:

Annual Income Range	Tax Rate
< $5,000	0% of total earned
$5,000 - $9,999	10% of total earned
$10,000 - $19,999	20% of total earned
>= $20,000	30% of total earned

Write a program to help calculate the tax for one person.

APPENDIX E. POSTTEST

IEDT 216 MIDTERM EXAM Fall 1994

Student Name:_____ SS#_____

I. [8 points] A C program contains the following prototypes:

```
int A( char a );
void B( int a, char b );
int C( int a );
float D( int a, int b, float c );
```

and variable declarations in the main function:

```
int a, b;
char c;
float d;
```

Indicate which of the following are correct C statements in the main function, and describe the errors for the incorrect statements.

	correct?(Yes or No)	if incorrect, why?
1. a = A(c);		
2. B(a, b, c);		
3. d = B(a, c);		
4. b =C(a);		
5. printf("%f",D(a,b,c));		

II. [4 points] What's the output of the following program.

```
#include<stdio.h>

void function1(void);
void function2(void);
void function3(void);

void main(void)
{    printf("dog\n");
     function1();
     function3();
}

void function1(void)
{   printf("cat\n");
}

void function2(void)
{   printf("fish\n");
    function1();
}

void function3(void)
{   printf("monkey\n");
    function2();
}
```

Answer:

III. [4 points] What's the output of the following program.

```
#include<stdio.h>

void increment( int a);

void main(void)
{  int a = 1;
   printf("a = %d\n", a);
   increment(a);
   printf("a = %d\n", a);
}

void increment( int a)
{  a++;
   printf("a = %d\n", a);
}
```

Answer:

IV. [4 points] What's the output of the following program.

```
#include<stdio.h>

void total( int i);

void main(void)
{  int i;

   for ( i = 1; i <= 3; i++)
      total (i);
}

void total( int i)
{   static int a = 0;
    int b = 0;

    a = a + i;
    b = b + i;
    printf("a = %d, b = %d\n", a, b);
}
```

Answer:

V. [15 points] The following program contains many repetitions of the same code. Rewrite the program by utilizing one function to avoid this situation.

```
#include<stdio.h>

void main(void)
{  int i,j;
   for(i=1; i <= 1; i++)
   {  for (j=1; j <= 1; j++)
        printf("*");
      printf("\n");
   }

   for(i=1; i <= 2; i++)
   {  for (j=1; j <= 2; j++)
        printf("*");
      printf("\n");
   }

   for(i=1; i <= 3; i++)
   {  for (j=1; j <= 3; j++)
        printf("*");
      printf("\n");
   }
}
```

Answer:

VI. [15 points] Please construct functions and write a C program that asks the user to enter an integer and prints a message to show if the number was positive or negative and if it was an odd or even number.

For example,

the integer	output message
0	zero
5	positive odd integer
10	positive even integer
-33	negative odd integer
-62	negative even integer

Answer:

APPENDIX F. MEANS OF THE POSTTEST FOR EACH QUESTION

Problem	Treatment	N	Means	SD	t-value	df	Prob. T
1	SEQUENCE1	6	4.67	1.51			
					0.27	13	0.79
	SEQUENCE2	9	4.44	1.59			
2	SEQUENCE1	6	4.00	0.00			
					1.79	8	0.11
	SEQUENCE2	9	3.22	1.30			
3	SEQUENCE1	6	2.17	0.98			
					-1.05	13	0.31
	SEQUENCE2	9	2.78	1.18			
4	SEQUENCE1	6	2.33	1.37			
					0.47	13	0.64
	SEQUENCE2	9	2.00	1.32			
5	SEQUENCE1	6	7.83	4.44			
					0.70	13	0.49
	SEQUENCE2	9	6.33	3.74			
6	SEQUENCE1	6	9.33	3.44			
					0.75	13	0.47
	SEQUENCE2	9	8.00	3.35			

APPENDIX G. INFORMATION SHEETS FOR LEARNING LESSON VINCENT

Obtaining a Vincent Account

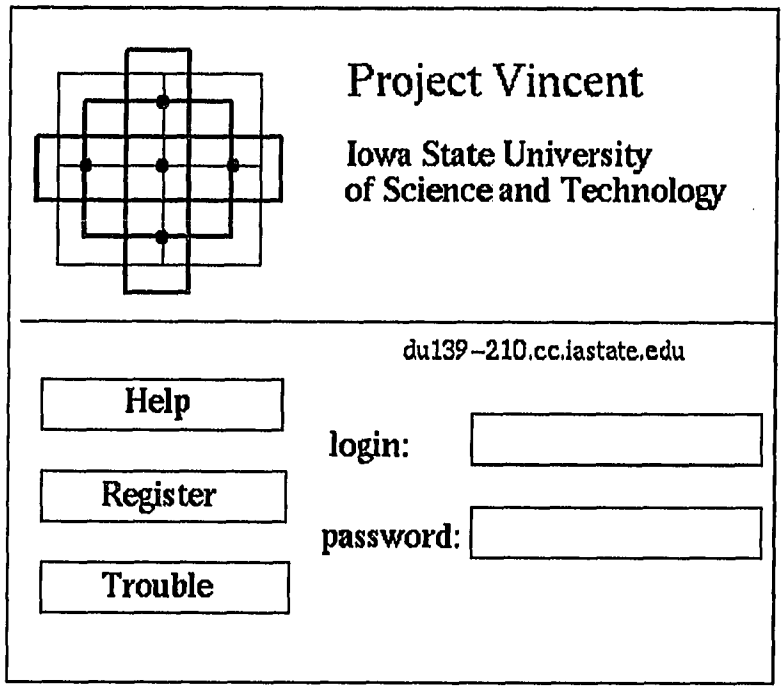


Figure G.1: Project Vincent screen

1. Position the mouse cursor over the Register box on the Vincent Screen(Figure G.1). Press the left mouse button.

2. After a few moments, a series of questions will be displayed. Answer each of the questions as they are asked. If all your answers match Registrar's office records, you will be asked to select your username. If you choose a username that is already in use, you will be asked to choose another. Your username may be up to 8 characters long and any combination of lowercase letters and numbers.

After you have chosen a username, you must choose a password. Passwords may be up to 16 characters long and can be composed of letters, numbers, and symbols. Notice that your password is not displayed on the screen. Therefore, you will be asked to enter the password twice.

After you have registered, please allow one or two days for your home directory and setup files to be created. After this time you should be able to login with your new username and password and use the Vincent system.

Note:

1. If at any point you wish to start over, press both the CTRL and c key at the same time.

2. If an error occurs during your registering, write down the message and contact the Access and Accounting office, 197 Durham Center (294-4171).

Login at a Vincent Workstation

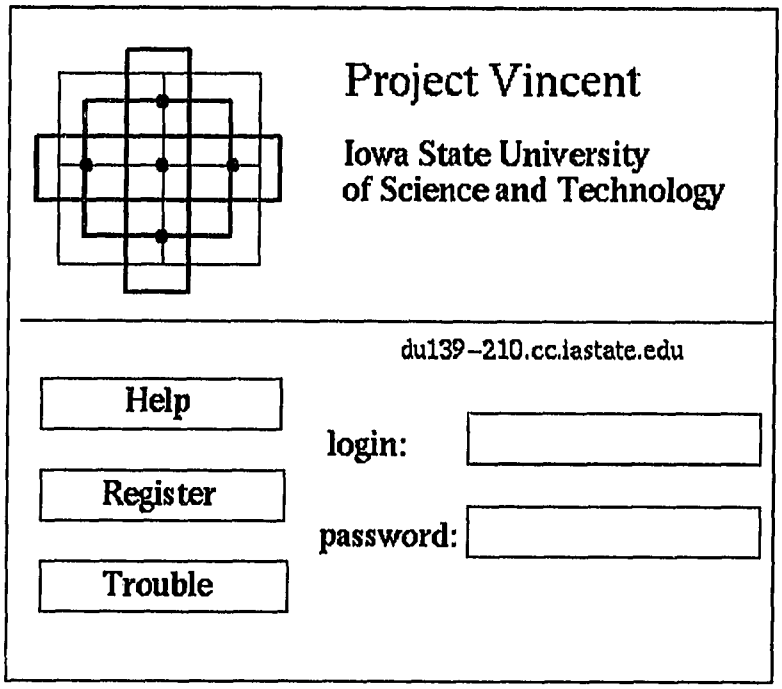


Figure G.2: Project Vincent screen

1. Position the mouse cursor over the login field on the Vincent Screen(Figure G.2). Enter your Project Vincent username and press [Return].

2. Enter your password at the password field and press [Return]. Notice that your password is not displayed on the screen.

If you provide correct information, you will login successfully. If you have a login error, a message box will appear. Click on [OK] to dismiss the message and repeat the login process or click on [Help] for further information.

Logout at a Vincent Workstation

1. Position the mouse cursor over the **Session** of the Dash menu and press the left mouse button. A pull down menu will be displayed on the screen (see Figure G.3).

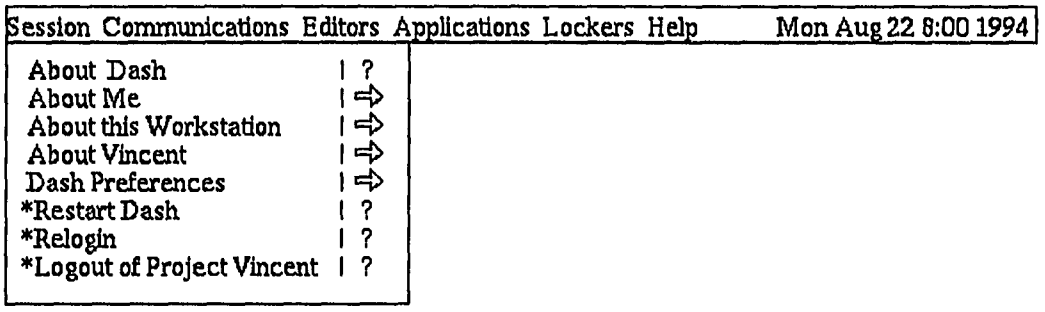


Figure G.3: Dash menu

2. Position the mouse cursor over the **Logout of Project Vincent** of the pull down menu of Session and press the left mouse button. The logout window will appear (see Figure G.4).

3. Position the mouse cursor over the [Yes] box and press the left mouse button. The confirmation window will appear (see Figure G.5).

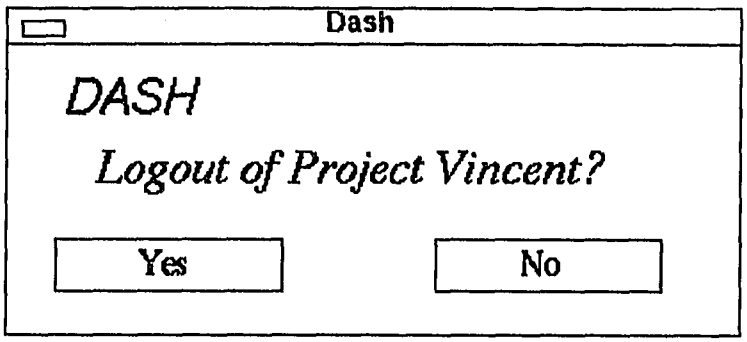


Figure G.4: Logout window

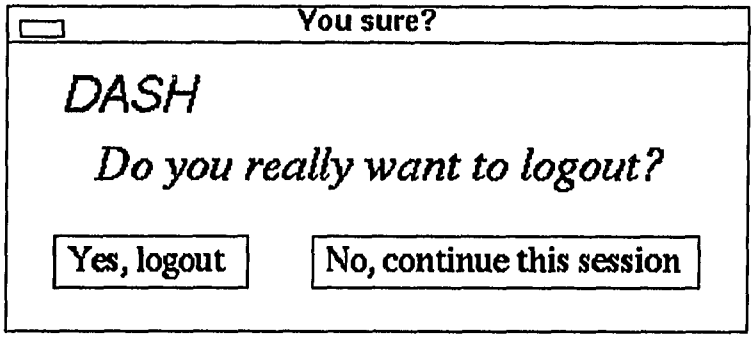


Figure G.5: Confirmation window

4. Position the mouse cursor over the Yes, logout box and press the left mouse button to logout.

Learning lesson vincent

When using lesson vincent at a **Project Vincent Workstation** you need to enter the following commands at the *vincent%* prompt:

vincent% **add abc** Return

vincent% **abcinit** Return *(first time only)*

vincent% **lesson vincent** Return

After a few moments, you will see the title page of the lesson vincent (Figure G.6).

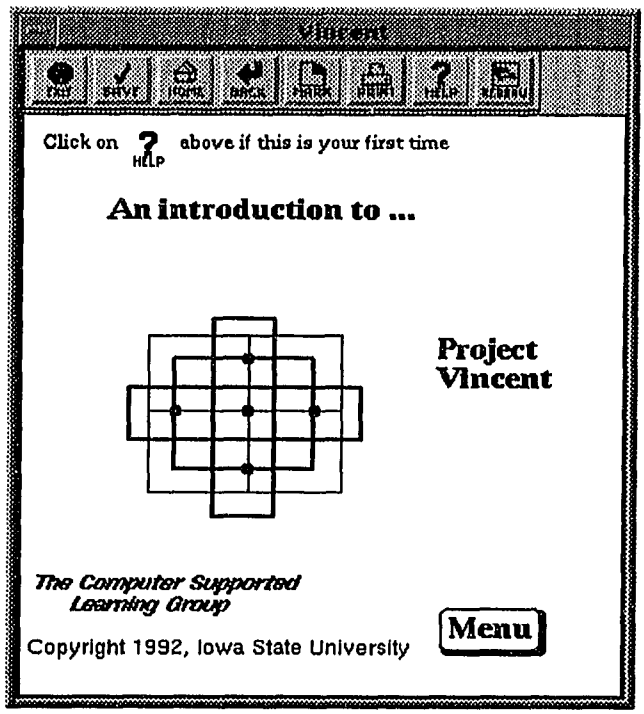


Figure G.6: Lesson Vincent title Page

Please position the mouse cursor over the Menu button and press the left mouse button to begin the lesson vincent.

If you wish, you can exit the lesson at any point by clicking on [EXIT] at the top of the lesson window. The exit window will appear (see Figure G.7).

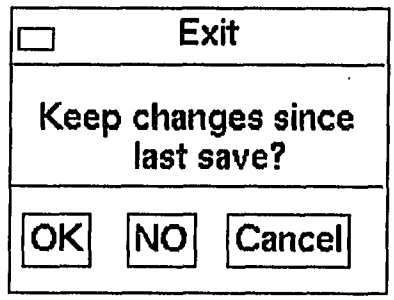


Figure G.7: Exit window

Click on [OK] to save and exit the lesson. Click on [NO] to exit the lesson without saving. Click on [Cancel] to cancel the exit action and continue to learn the lesson.

After you exit the lesson if you want to learn the lesson again, just enter the following command:

vincent% **lesson vincent** [**Return**]

Note:

1. The command **add abc** allows you to attach the abc locker. You will need to do this command once each time you login .

2. If something unexpected happen, and you have trouble exiting the lesson normally, just click on the window where you typed **lesson vincent** and press both the [CTRL] and [c] key at the same time to exit.

APPENDIX H. INFORMATION SHEETS FOR LEARNING LESSON FUNCTION

Learning lesson function

When using lesson function at a **Project Vincent Workstation** you need to enter the following commands at the *vincent%* prompt:

vincent% **add fun** [Return]

vincent% **add abc** [Return]

vincent% **abcinit** [Return] *(first time only)*

vincent% **lesson function** [Return]

After a few moments, you will see the title page of the lesson function (Figure H.1).

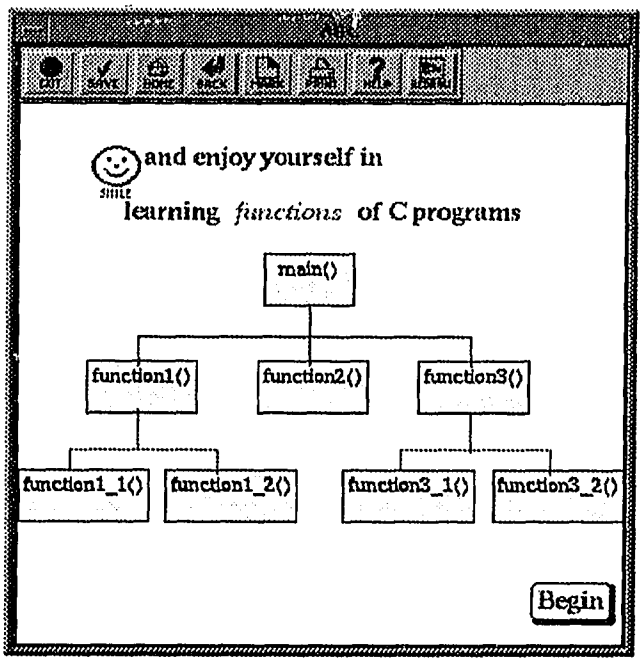


Figure H.1: Lesson function title Page

Please position the mouse cursor over the [Begin] button and press the left mouse button to begin the lesson function.

If you wish, you can exit the lesson at any point by clicking on [EXIT] at the top of the lesson window. The exit window will appear (see Figure H.2).

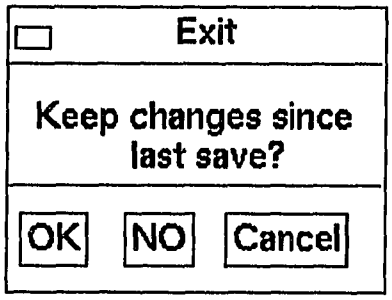


Figure H.2: Exit window

Click on [OK] to save and exit the lesson. Click on [NO] to exit the lesson without saving. Click on [Cancel] to cancel the exit action and continue to learn the lesson.

After you exit the lesson if you want to learn the lesson again, just enter the following command:

vincent% **lesson function** [**Return**]

Note:

1. The commands **add fun** and **add abc** allow you to attach the fun and abc lockers. You will need to do these commands once each time you login.

2. If something unexpected happen, and you have trouble in exiting the lesson normally, just click on the window where you typed **lesson function** and press both the [CTRL] and [c] key at the same time to exit.